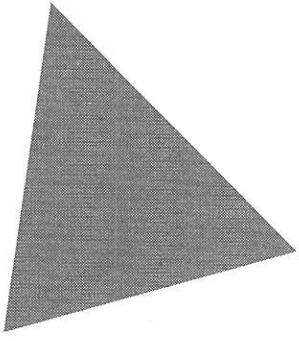
Object Pascal Language Guide

Borland®

Delphi™ 3

for Windows 95 & Windows NT

Object Pascal Language Guide



Borland® Delphi™ for Windows 95 and Windows NT

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 1997 Borland International. All rights reserved. All Borland product names are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

HDA1330WW21771 1E0R0297

9798990001-9 8 7 6 5 4 3 2 1

H1

Contents

Chapter 1
Introduction 1-1
What's in this manual? . . . 1-1
Syntax diagrams . . . 1-2
Manual conventions . . . 1-2
Software registration and technical support . . 1-3

Chapter 2
Tokens 2-1
Special symbols . . . 2-1
Reserved words and standard directives 2-2
Identifiers . . . 2-3
Numbers . . . 2-4
Labels . . . 2-5
Character strings . . . 2-5
Comments . . . 2-6

Chapter 3
Constants 3-1
Resource strings . . . 3-2

Chapter 4
Types 4-1
Fundamental and generic types . . . 4-1
Simple types . . . 4-2
Ordinal types . . . 4-2
Integer types . . . 4-3
Character types . . . 4-4
Enumerated types . . . 4-5
Boolean types . . . 4-5
Subrange types . . . 4-6
Real types . . . 4-6
String types . . . 4-7
Short string types . . . 4-8
Long string types . . . 4-8
Wide string types . . . 4-11
Using strings . . . 4-11
Structured types . . . 4-12
Array types . . . 4-12
Record types . . . 4-13
Set types . . . 4-15
File types . . . 4-15
Pointer types . . . 4-16
Type Pointer . . . 4-16
Character pointers . . . 4-16
Standard pointers . . . 4-17
Procedural types . . . 4-18
Global procedure pointers . . . 4-18
Method pointers . . . 4-18
Procedural values . . . 4-19
Procedural type compatibility . . . 4-20
Variant types . . . 4-20
Values in variants . . . 4-21
Variant type conversions . . . 4-22
Variant expressions . . . 4-25
OleVariant . . . 4-26
Variant arrays . . . 4-26
Variants and OLE Automation objects . . . 4-29
Identical and compatible types . . . 4-31
Type identity . . . 4-31
Type compatibility . . . 4-32
Assignment compatibility . . . 4-33
The type declaration part . . . 4-34

Chapter 5
Variables and typed constants 5-1
Variable declarations . . . 5-1
Local variables and the stack . . . 5-2
Variable references . . . 5-2
Qualifiers . . . 5-3
Arrays, strings, and indexes . . . 5-3
Records and field designators . . . 5-4
Object component designators . . . 5-5
Pointers and dynamic variables . . . 5-5
Variable typecasts . . . 5-5
Typed constants . . . 5-7
Simple-type constants . . . 5-7
String-type constants . . . 5-8
Structured-type constants . . . 5-8
Array-type constants . . . 5-8
Record-type constants . . . 5-9
Set-type constants . . . 5-10
Pointer-type constants . . . 5-10
Procedural-type constants . . . 5-11

Chapter 6
Expressions 6-1
Expression syntax . . . 6-2
Operators . . . 6-4
Arithmetic operators . . . 6-4
Logical operators . . . 6-5

Boolean operators 6-6
String operator 6-6
Character-pointer operators 6-7
Set operators 6-7
Variant operators 6-8
Relational operators 6-8
Comparing simple types 6-8
Comparing strings 6-8
Comparing packed strings 6-9
Comparing pointers and references . . . 6-9
Comparing character pointers 6-9
Comparing variants 6-9
Comparing sets 6-9
Testing set membership 6-9
Class operators 6-9
The @ operator 6-10
@ with a variable 6-10
@ with a procedure, function, or method 6-10
Function calls 6-10
Set constructors 6-11
Value typecasts 6-12
Procedural types in expressions 6-12

Chapter 7
Statements **7-1**
Simple statements 7-1
Assignment statements 7-1
Procedure statements 7-2
Goto statements 7-2
Structured statements 7-3
Compound statements 7-3
Conditional statements 7-3
If statements 7-4
Case statements 7-4
Repetitive statements 7-5
Repeat statements 7-6
While statements 7-6
For statements 7-7
With statements 7-9
Exception statements 7-10

Chapter 8
Blocks, locality, and scope **8-1**
Blocks 8-1
Rules of scope 8-3
Block scope 8-3
Record scope 8-3
Class scope 8-3
Unit scope 8-4

Chapter 9
Procedures and functions **9-1**
Procedure declarations 9-1
Calling conventions 9-2
Forward declarations 9-3
External declarations 9-4
Assembler declarations 9-5
Function declarations 9-5
Parameters 9-6
Value parameters 9-7
Constant parameters 9-7
Variable parameters 9-8
Out parameters 9-8
Untyped parameters 9-8
Open-array parameters 9-9
Open-array constructors 9-11
Type variant open-array parameters 9-11

Chapter 10
Class types **10-1**
Instances and references 10-2
Class components 10-3
Fields 10-3
Methods 10-3
Properties 10-4
Inheritance 10-4
Components and scope 10-4
Forward references 10-4
Class type compatibility rules 10-5
Component visibility 10-5
Public components 10-6
Published components 10-6
Automated components 10-6
Protected components 10-7
Private components 10-7
Static methods 10-8
Virtual methods 10-8
Dynamic methods 10-9
Abstract methods 10-10
Method activations 10-10
Method implementations 10-10
Constructors and destructors 10-12
Constructors 10-12
Destructors 10-14
Class operators 10-15
The is operator 10-15

The as operator 10-16
Message handling 10-16
Message handler declarations. 10-16
Message handler implementations 10-17
Message dispatching 10-17
Property syntax 10-18
Property definitions 10-19
Property access 10-20
Access specifiers 10-21
Array properties 10-21
Index specifiers 10-23
Storage specifiers 10-24
Property overrides 10-25
Class-reference types 10-25
Constructors and class references 10-27
Class methods 10-28
The TObject and TClass types 10-28

Chapter 11
Exceptions **11-1**
Using exception handling 11-1
Exception declarations 11-2
The raise statement 11-2
The try...except statement 11-3
Re-raising exceptions 11-6
Nested exceptions 11-6
The try...finally statement 11-7
Exit, Break, and Continue procedures 11-8
Predefined exceptions 11-8
Exception handling support routines 11-9

Chapter 12
Programs and units **12-1**
Program syntax 12-1
The program heading 12-1
The uses clause 12-2
Unit syntax 12-2
The unit heading 12-2
The interface part. 12-3
The implementation part 12-3
The initialization part 12-3
The finalization part 12-4
Indirect unit references 12-4
Circular unit references 12-5

Chapter 13
Interfaces **13-1**
Interface syntax 13-2
IUnknown and inheritance 13-3
Interface identification 13-4
Calling conventions 13-5
Member visibility 13-5
Access specifiers 13-5
Forward declarations 13-5
Interface implementations 13-6
Class declaration 13-6
Interface method implementation 13-7
Mapping interface methods 13-7
Method resolution clauses 13-8
Method resolution clause restrictions 13-9
Interface references 13-11
Reference counting 13-11
Type compatibility 13-11
Interface conversions 13-12
Using assignment to obtain an interface reference 13-12
Interface querying 13-13
Interactions with other language features 13-14
OLE automation Interface types. 13-14
Dispatch interfaces 13-14
Dual interfaces. 13-16

Chapter 14
Dynamic-link libraries **14-1**
What is a DLL? 14-1
Using DLLs 14-2
Import units 14-3
Static and dynamic imports 14-4
Writing DLLs 14-5
The exports clause. 14-6
Library initialization code 14-7
Library programming notes 14-8
Global variables in a DLL 14-8
DLLs and the System unit 14-8
Exceptions and runtime errors in DLLs. . . 14-8
The Shared Memory Manager 14-9
The DLLProc variable. 14-9

Chapter 15
Packages **15-1**
Package Source Files 15-1
Naming Packages 15-2
The requires clause 15-2
Avoiding circular package references 15-3
Duplicate package references. 15-3

The contains clause 15-3
Avoiding redundant source code uses 15-3
Compiling packages 15-3
Generated files 15-4
Package-specific compiler directives 15-4
Package-specific command-line compiler switches 15-5

Chapter 16
Input and output **16-1**
File input and output 16-2
Text files 16-3
Untyped files 16-4
Text-file device drivers 16-4
The Open function 16-5
The InOut function 16-5
The Flush function 16-5
The Close function 16-6

Chapter 17
Using null-terminated strings **17-1**
What is a null-terminated string? 17-1
Using null-terminated strings 17-1
Character pointers and string literals 17-2
Character pointers and character arrays . . . 17-2
Character pointer indexing 17-3
Null-terminated strings and standard procedures 17-4
Null-terminated string functions 17-5
Mixing long strings and null-terminated strings 17-5
Null-terminated wide character strings 17-6

Chapter 18
Memory issues **18-1**
Windows memory management 18-1
The memory manager 18-1
Internal data formats 18-2
Integer types 18-2
Character types 18-2
Boolean types 18-2
Enumerated types 18-3
Floating-point types 18-3
The Real type 18-3
The Single type 18-3
The Double type 18-4
The Extended type 18-4
The Comp type 18-4
The Currency type 18-4
Pointer types 18-5
Short string types 18-5
Long string types 18-5
Wide string types 18-6
Set types 18-6
Array types 18-6
Record types 18-7
File types 18-7
Procedural types 18-8
Class types 18-9
Class reference types 18-10
Variant types 18-10

Chapter 19
Control issues **19-1**
Calling conventions 19-1
Parameters 19-2
Function results 19-3
Method calling conventions 19-4
Constructors and destructors 19-4
Register saving conventions 19-5
Exit procedures 19-5

Chapter 20
The built-in assembler **20-1**
The asm statement 20-1
Register use 20-2
Assembler statement syntax 20-2
Labels 20-2
Instruction opcodes 20-3
RET instruction sizing 20-3
Automatic jump sizing 20-3
Assembler directives 20-4
Operands 20-5
Expressions 20-6
Differences between Object Pascal and Assembler expressions 20-6
Expression elements 20-7
Constants 20-7
Symbols 20-9
Expression classes 20-11
Expression types 20-13
Expression operators 20-14
Assembler procedures and functions 20-16

Tables

1.1 Typefaces and symbols in these manuals. 1-2
2.1 Object Pascal reserved words. 2-2
2.2 Object Pascal directives 2-3
4.1 Fundamental integer types 4-3
4.2 Generic integer types for 16-bit implementations of Object Pascal 4-3
4.3 Generic integer types for 32-bit implementations of Object Pascal 4-3
4.4 Real data types 4-7
4.5 Some pointers declared in Delphi 4-17
4.6 Variant type constants and values 4-21
4.7 Expression types and resulting variant type codes 4-22
4.8 Converting a variant to an integer type value 4-23
4.9 Converting a variant to a real type value 4-23
4.10 Converting a variant to a date and time value 4-23
4.11 Converting a variant to a string type value 4-24
4.12 Converting a variant to a Boolean type value 4-24
4.13 Variant operation type matrix 4-25
4.14 Non-relational variant operator results . . . 4-26
6.1 Precedence of operators 6-1
6.2 Binary arithmetic operations 6-4
6.3 Unary arithmetic operations 6-4
6.4 Logical operations 6-5
6.5 Boolean operations 6-6
6.6 String operation 6-6
6.7 Permitted PChar constructs 6-7
6.8 Set operations. 6-7
6.9 Relational operations 6-8
9.1 Summary of calling conventions 9-3
9.2 Type variant open-array expressions. . . . 9-13
15.1 Compiled package files. 15-4
15.2 Package-specific compiler directives 15-4
15.3 Package-specific command-line compiler switches 15-5
16.1 Input and output procedures and functions. 16-1
17.1 Null-terminated string functions 17-5
18.1 Long string dynamic memory layout . . . 18-5
18.2 Wide string dynamic memory layout . . . 18-6
18.3 Type alignment masks 18-7
18.4 Virtual Method Table layout. 18-9
19.1 Summary of calling conventions 19-1
20.1 Built-in assembler reserved words 20-5
20.2 String examples and their values 20-8
20.3 CPU registers. 20-8
20.4 Values, classes, and types of symbols . . . 20-10
20.5 Predefined type symbols. 20-14
20.6 Summary of built-in asssembler expression operators 20-14
20.7 Definitions of built-in assembler expression operators 20-15

Chapter

1

Introduction

This manual is about the Object Pascal language as it is used in Delphi. For an overview of the Delphi documentation, see the introduction to the *User's Guide*. This manual:

- Presents the formal definition of the Object Pascal language
- Describes what goes on inside an Object Pascal application in regard to memory, data formats, calling conventions, input and output, and automatic optimizations

You can also find information about these and other topics in online Help.

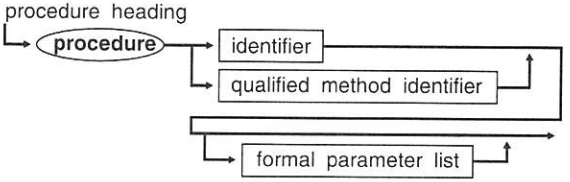
What's in this manual?

This manual contains several different kinds of information:

- The first thirteen chapters define the Object Pascal language, examining each element of a program in detail.
- The next seven chapters present technical information for advanced users about input and output, numerical processing, memory management, program control, and optimization.

Syntax diagrams

In this book you'll encounter *syntax diagrams*. For example:



To read a syntax diagram, follow the arrows. Frequently, more than one path is possible. The above diagram indicates that a formal parameter list is optional in a procedure heading. You can follow the path from the identifier to the end of the procedure heading, or you can follow it to the formal parameter list before reaching the end.

The names in the boxes stand for constructions. Those in circles—reserved words. operators, and punctuation—are actual terms used in the programs; they are boldfaced in the diagrams.

Manual conventions

The printed manuals for Delphi use the typefaces and symbols described in Table 1.1 to indicate special text.

Table 1.1 Typefaces and symbols in these manuals

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears on screen or in Object Pascal code. It also represents anything you must type.
[ ]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
Boldface	Boldfaced words in text or code listings represent Object Pascal reserved words or compiler options.
Italics	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."

Software registration and technical support

The Borland Assist program offers a range of technical support plans to fit the diverse needs of individuals, consultants, large corporations, and developers. To receive help with this product, return the registration card and select the Borland Assist plan that best suits your needs. North American customers can register by

phone 24 hours a day at 1-800-845-0147. For additional details on these and other Borland services, see the *Borland Assist Support and Services Guide* included with this product.

Chapter

2

Tokens

Tokens are the smallest meaningful units of text in an Object Pascal program. They are categorized as special symbols, identifiers, labels, numbers, and string constants.

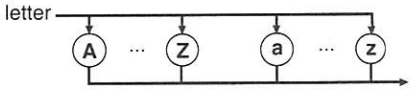
An Object Pascal program is made up of tokens and *separators*. A separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators if each token is a reserved word, an identifier, a label, or a number. Separators can't be part of tokens except in string constants.

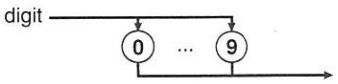
Special symbols

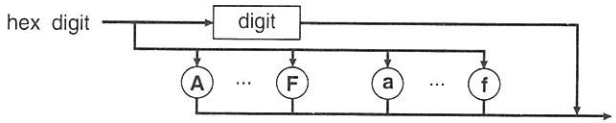
Object Pascal uses the following subsets of the ASCII character set:

- **Letters**—the English alphabet, *A* through *Z* and *a* through *z*
- **Digits**—the Arabic numerals 0 through 9
- **Hex digits**—the Arabic numerals 0 through 9, the letters *A* through *F*, and the letters *a* through *f*
- **Blanks**—the space character (ASCII 32) and all ASCII control characters (ASCII 0 through 31), including the end-of-line or return character (ASCII 13)

These are the syntax diagrams for letter, digit, and hex digit:







Special symbols and reserved words are characters that have one or more fixed meanings. The following single characters are special symbols:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { } $ #
```

These character pairs are also special symbols:

```
<= >= := .. (* *) (. .) //
```

A left bracket—[—is equivalent to the character pair of left parenthesis and period—(.—and a right bracket—]—is equivalent to the character pair of period and right parenthesis—.). Likewise, a left brace—{—is equivalent to the character pair of left parenthesis and asterisk—(*—and a right brace—}—is equivalent to the character pair of asterisk and right parenthesis—*).

Reserved words and standard directives

Reserved words can't be redefined.

Reserved words appear in lowercase **boldface** throughout this manual. Object Pascal is *not* case sensitive, however, so you can use either uppercase or lowercase letters in your programs.

Table 2.1 shows Object Pascal's reserved words:

Table 2.1 Object Pascal reserved words

and	downto	in	out	stringresource
array	else	inherited	packed	then
as	end	initialization	procedure	threadvar
asm	except	inline	program	to
begin	exports	interface	property	try
case	file	is	raise	type
class	finalization	label	record	unit
const	finally	library	repeat	until
constructor	for	mod	resourcestring	uses
destructor	function	nil	set	var
dispinterface	goto	not	shl	while
div	if	object	shr	with
do	implementation	of	string	xor
		or		

Table 2.2 shows Object Pascal's standard directives. Directives are used only in contexts where user-defined identifiers can't occur. Unlike reserved words, you can redefine standard directives, but we advise that you don't.

Table 2.2 Object Pascal directives

absolute	dynamic	name	public	safecall
abstract	export	near	published	stdcall
assembler	external	nodefault	read	stored
automated	far	override	readonly	virtual
cdecl	forward	pascal	register	write
default	index	private	resident	writeonly
dispid	message	protected		

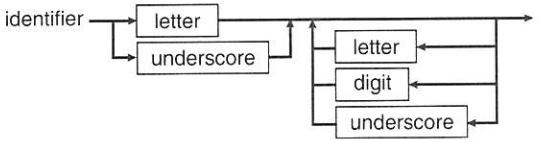
Private, protected, public, published, and **automated** act as reserved words within object type declarations, but are otherwise treated as directives.

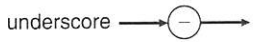
Identifiers

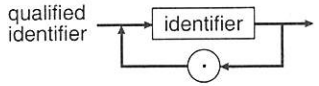
Identifiers denote constants, types, variables, procedures, functions, units, programs, and fields in records.

An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with a letter or an underscore character (_) and can't contain spaces. Letters, digits, and underscore characters (ASCII $5F) are allowed after the first character. Like reserved words, identifiers are *not* case sensitive.

When several instances of the same identifier exist, you may need to qualify the identifier by another identifier to select a specific instance. For example, to qualify the identifier *Ident* by the unit identifier *UnitName*, write *UnitName.Ident*. The combined identifier is called a *qualified identifier*. Units are described in Chapter 6 of the *User's Guide* and Chapter 12 of this manual.







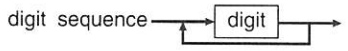
Here are some examples of identifiers and qualified identifiers:

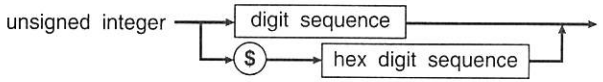
```
Writeln
Exit
Real2String
System.Close
SysUtils.StrLen
```

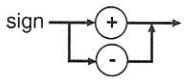
In this manual, standard and user-defined identifiers are *italicized* when they are referred to in text.

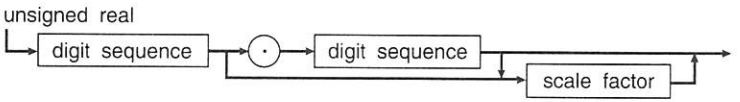
Numbers

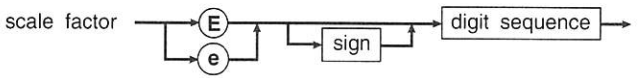
Ordinary decimal notation is used for numbers that are constants of integer and real types. A hexadecimal integer constant uses a dollar sign ($) as a prefix. Engineering notation (E or e, followed by an exponent) is read as "times ten to the power of" in real types. For example, 7E–2 means 7×10^{-2}; 12.25e+6 or 12.25e6 both mean $12.25 \times 10^{+6}$. Syntax diagrams for writing numbers follow:

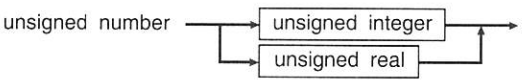


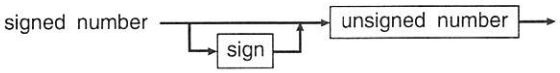










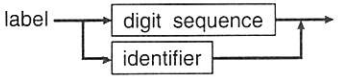


Numbers with decimals or exponents denote real-type constants. Other decimal numbers denote integer-type constants; they must be within the range –2,147,483,648 to 2,147,483,647.

Hexadecimal numbers denote integer-type constants; they must be within the range $00000000 to $FFFFFFFF. The resulting value's sign is implied by the hexadecimal notation.

Labels

A label is a digit sequence in the range 0 to 9999. Leading zeros are not significant. Labels are used with **goto** statements.



As an extension to Standard Pascal, Object Pascal allows identifiers to function as labels.

Character strings

A character string is a sequence of zero or more characters from the extended ASCII character set, written on one line in the program and enclosed by apostrophes.

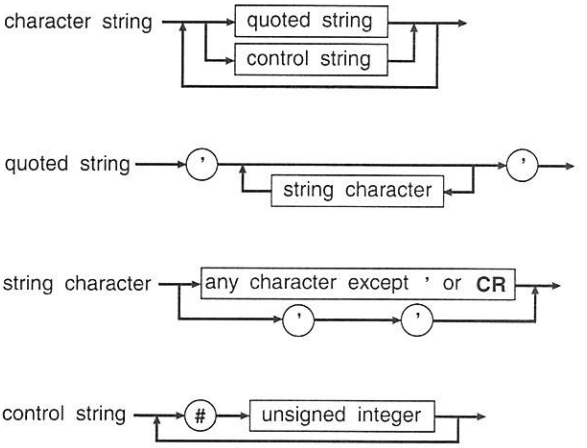
A character string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a character string denote a single character, an apostrophe. For example,

```
'BORLAND'           { BORLAND }
'You''ll see'       { You'll see }
''''                { ' }
''                  { null string }
' '                 { a space }
```

Object Pascal lets you embed control characters in character strings. The # character followed by an unsigned integer constant in the range 0 to 255 denotes a character of the corresponding ASCII value. There must be no separators between the # character and the integer constant. Likewise, if several are part of a character string, there must be no separators between them. For example,

```
#13#10
'Line 1'#13'Line2'
```

```
#7#7'Wake up!'#7#7
```



A character string's *length* is the actual number of characters in the string. A character string of any length is compatible with any string type, and with the *PChar* type when the extended syntax is enabled {**$X+**}. Also, a character string of length one is compatible with any *Char* type, and a character string of length *N*, where *N* is greater than or equal to one, is compatible with packed arrays of *N* characters.

Comments

The following constructs are comments and are ignored by the compiler:

```
{ Any text not containing right brace }
(* Any text not containing star/right parenthesis *)
// Any text from a double-slash to the end of the line
```

A comment that contains a dollar sign (**$**) immediately after the opening { or (* is a *compiler directive*. A mnemonic of the compiler command follows the **$** character.

Chapter

3

Constants

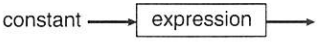
A constant is an identifier that marks a value that can't change. A *constant declaration* declares a constant within the block containing the declaration. A constant identifier can't be included in its own declaration.

Object Pascal allows the use of *constant expressions*. A constant expression is an expression that can be evaluated by the compiler without actually executing the program. Wherever Standard Pascal allows only a simple constant, Object Pascal allows a constant expression.

Examples of constant expressions follow:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Pascal'
Chr(32)
Ord('Z') - Ord('A') + 1
```

The simplest case of a constant expression is a simple constant, such as 100 or 'A'.



Because the compiler has to be able to completely evaluate a constant expression at compile time, the following constructs are not allowed in constant expressions:

- References to variables and typed constants (except in constant address expressions as described on page 5-7)
- Function calls (except those noted in the following text)
- The address operator (@) (except in constant address expressions as described on page 5-7)

Except for these restrictions, constant expressions follow the syntactical rules as ordinary expressions. For expression syntax, see Chapter 6, "Expressions."

The following standard functions are allowed in constant expressions:

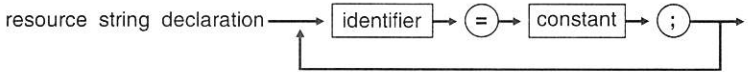
Abs	High	Odd	*SizeOf*
Addr	*Length*	Ord	*Succ*
Chr	Lo	Pred	*Swap*
Hi	Low	Round	*Trunc*

Here are some examples of the use of constant expressions in constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Resource strings

A *Resource String* is a constant that is automatically stored in a resource and compiled into the executable, DLL, or package file. The compiler automatically handles collisions of duplicate resource strings contained in separate packages. Another resource string cannot be included in a resource string declaration.



Here are some examples of the use of resource string expressions in resource string declarations:

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';

  CompanyName = 'Borland International\000\000';
  ProductName = 'Borland Delphi\000\000';
```

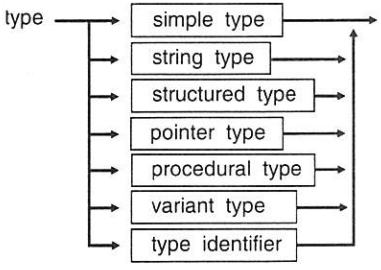
Chapter

4

Types

When you declare a variable, you must state its *type*. A variable's type circumscribes the set of values it can have and the operations that can be performed on it. A *type declaration* specifies the identifier that denotes a type.

When an identifier occurs on the left side of a type declaration, it's declared as a *type identifier* for the block in which the type declaration occurs. A type identifier's scope doesn't include itself except for pointer types.



There are seven major type classes. They are described in the following sections.

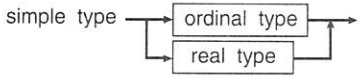
Fundamental and generic types

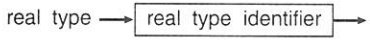
Object Pascal's predefined types are divided into two categories: *fundamental* types and *generic* types. The range and format of the fundamental types is independent of the underlying CPU and operating system and does not change across different implementations of Object Pascal. The range and format of the generic types, on the other hand, depends on the underlying CPU and operating system.

There are currently three classes of predefined types that distinguish between fundamental and generic types: integer types, character types, and string types. For all other classes, you should regard the predefined types as fundamental.

Simple types

Simple types define ordered sets of values.





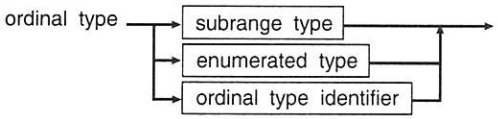
A real type identifier is one of the standard identifiers: *Real*, *Single*, *Double*, *Extended*, *Currency* or *Comp*. Chapter 2, "Tokens," explains how to denote constant integer type and real type values.

Ordinal types

Ordinal types are a subset of simple types. All simple types other than real types are ordinal types, which are set off by six characteristics:

- All possible values of a given ordinal type are an ordered set, and each possible value is associated with an *ordinality*, which is an integral value. Except for integer type values, the first value of every ordinal type has ordinality 0, the next has ordinality 1, and so on for each value in that ordinal type. The ordinality of an integer type value is the value itself. In any ordinal type, each value other than the first has a predecessor, and each value other than the last has a successor based on the ordering of the type.
- The standard function *Ord* can be applied to any ordinal type value to return the ordinality of the value.
- The standard function *Pred* can be applied to any ordinal type value to return the predecessor of the value. If applied to the first value in the ordinal type and if range checking is enabled {**$R+**}, *Pred* produces a runtime error.
- The standard function *Succ* can be applied to any ordinal type value to return the successor of the value. If applied to the last value in the ordinal type and if range checking is enabled {**$R+**}, *Succ* produces a runtime error.
- The standard function *Low* can be applied to an ordinal type identifier and to a variable reference of an ordinal type. The result is the lowest value in the range of the given ordinal type.
- The standard function *High* can be applied to an ordinal type identifier and to a variable reference of an ordinal type. The result is the highest value in the range of the given ordinal type.

The syntax of an ordinal type follows:



Object Pascal has twelve predefined ordinal types: *Integer*, *Shortint*, *Smallint*, *Longint*, *Byte*, *Word*, *Cardinal*, *Boolean*, *ByteBool*, *WordBool*, *LongBool*, and *Char*. There are also two other classes of user-defined ordinal types: enumerated types and subrange types.

Integer types

Object Pascal's predefined integer types are divided into two categories: *fundamental* types and *generic* types. The difference between fundamental and generic types is explained on page 4-1.

The fundamental integer types are *Shortint*, *Smallint*, *Longint*, *Byte*, and *Word*. Each fundamental integer type denotes a specific subset of the whole numbers, according to the following table:

Table 4.1 Fundamental integer types

Type	Range	Format
Shortint	–128..127	Signed 8-bit
Smallint	–32768..32767	Signed 16-bit
Longint	–2147483648..2147483647	Signed 32-bit
Byte	0..255	Unsigned 8-bit
Word	0..65535	Unsigned 16-bit

The generic integer types are *Integer* and *Cardinal*. The *Integer* type represents a generic signed integer, and the *Cardinal* type represents a generic unsigned integer. The actual ranges and storage formats of the *Integer* and *Cardinal* vary across different implementations of Object Pascal, but are generally the ones that result in the most efficient integer operations for the underlying CPU and operating system.

Table 4.2 Generic integer types for 16-bit implementations of Object Pascal

Type	Range	Format
Integer	–32768..32767	Signed 16-bit
Cardinal	0..65535	Unsigned 16-bit

Table 4.3 Generic integer types for 32-bit implementations of Object Pascal

Type	Range	Format
Integer	–2147483648..2147483647	Signed 32-bit
Cardinal	0..2147483647	Unsigned 32-bit

Applications should use the generic integer formats whenever possible, since they generally result in the best performance for the underlying CPU and operating system. The fundamental integer types should be used only when the actual range and/or storage format matters to the application.

Arithmetic operations with integer-type operands use 8-bit, 16-bit, or 32-bit precision, according to the following rules:

- The type of an integer constant is the predefined integer type with the smallest range that includes the value of the integer constant.
- For a binary operator (an operator that takes two operands), both operands are converted to their common type before the operation. The common type is the predefined integer type with the smallest range that includes all possible values of both types. For example, the common type of *Smallint* and *Byte* is *Smallint*, and the common type of *Smallint* and *Word* is *Longint*. The operation is performed using the precision of the common type, and the result type is the common type.
- The expression on the right of an assignment statement is evaluated independently from the size or type of the variable on the left.
- Any byte-sized operand is converted to an intermediate word-sized operand that is compatible with both *Smallint* and *Word* before any arithmetic operation is performed.

An integer-type value can be explicitly converted to another integer type through typecasting. Typecasting is described in Chapter 5, "Variables and typed constants," and Chapter 6, "Expressions."

Character types

Object Pascal defines two fundamental character types and a generic character type. The difference between fundamental and generic types is explained on page 4-1.

The fundamental character types are *AnsiChar* and *WideChar*.

AnsiChar's values are byte-sized characters, ordered according to the extended ANSI character set.

WideChar's values are word-sized characters, ordered according to the Unicode character set. The first 256 Unicode characters correspond to the ANSI characters.

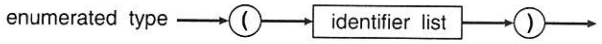
The generic character type is *Char*. In the current implementation of Object Pascal, *Char* corresponds to the type *AnsiChar*, but implementations for other CPUs and operating systems might define *Char* to be *WideChar*. When writing code that might need to handle characters of either size, use the standard function *SizeOf* instead of a hard-coded constant for character size.

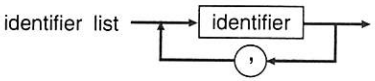
The function call *Ord(Ch)*, where *Ch* is any character-type value, returns *Ch*'s ordinality.

A string constant of length 1 can denote a constant character value. Any character value can be generated with the standard function *Chr*.

Enumerated types

Enumerated types define ordered sets of values by enumerating the identifiers that denote these values. Their ordering follows the sequence the identifiers are enumerated in.





When an identifier occurs within the identifier list of an enumerated type, it's declared as a constant for the block the enumerated type is declared in. This constant's type is the enumerated type being declared.

An enumerated constant's ordinality is determined by its position in the identifier list it's declared in. The enumerated type it's declared in becomes the constant's type. The first enumerated constant in a list has an ordinality of zero.

Here's an example of an enumerated type:

```
type
  Suit = (Club, Diamond, Heart, Spade);
```

Given these declarations, *Diamond* is a constant of type *Suit*.

When the *Ord* function is applied to an enumerated type's value, *Ord* returns an integer that shows where the value falls with respect to the other values of the enumerated type. Given the preceding declarations, *Ord(Club)* returns zero, *Ord(Diamond)* returns 1, and so on.

Boolean types

There are four predefined Boolean types: *Boolean*, *ByteBool*, *WordBool*, and *LongBool*. Boolean values are denoted by the predefined constant identifiers *False* and *True*. Because Booleans are enumerated types, these relationships hold:

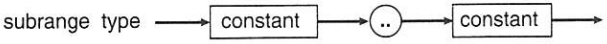
Boolean	**ByteBool, WordBool, LongBool**
False < *True*	*False* <> *True*
Ord(*False*) = 0	*Ord*(*False*) = 0
Ord(*True*) = 1	*Ord*(*True*) = any non-zero value
Succ(*False*) = *True*	*Succ*(*False*) = *True*
Pred(*False*) = *True*	*Pred*(*False*) = *True*

Boolean and *ByteBool* variables occupy one byte, a *WordBool* variable occupies two bytes (one word), and a *LongBool* variable occupies four bytes (two words). *Boolean* is the preferred type and uses the least memory; *ByteBool*, *WordBool*, and *LongBool* primarily exist to provide compatibility with other languages and the Windows environment.

A *Boolean* variable can assume the ordinal values 0 and 1 only, but variables of type *ByteBool*, *WordBool*, and *LongBool* can assume other ordinal values. An expression of type *ByteBool*, *WordBool*, or *LongBool* is considered *False* when its ordinal value is zero, and *True* when its ordinal value is nonzero. Whenever a *ByteBool*, *WordBool*, or *LongBool* value is used in a context where a *Boolean* value is expected, the compiler will automatically generate code that converts any nonzero value to the value *True*.

Subrange types

A subrange type is a range of values from an ordinal type called the *host type*. The definition of a subrange type specifies the smallest and the largest value in the subrange; its syntax follows:



Both constants must be of the same ordinal type. Subrange types of the form *A..B* require that *A* is less than or equal to *B*.

These are examples of subrange types:

```
0..99
-128..127
Club..Heart
```

A variable of a subrange type has all the properties of variables of the host type, but its runtime value must be in the specified interval.

One syntactic ambiguity arises from allowing constant expressions where Standard Pascal only allows simple constants. Consider the following declarations:

```
const
  X = 50;
  Y = 10;
type
  Color = (Red, Green, Blue);
  Scale = (X - Y) * 2..(X + Y) * 2;
```

Standard Pascal syntax dictates that, if a type definition starts with a parenthesis, it's an enumerated type, such as the *Color* type in the previous example. The intent of the declaration of *scale* is to define a subrange type, however. The solution is to reorganize the first subrange expression so that it doesn't start with a parenthesis, or to set another constant equal to the value of the expression and use that constant in the type definition:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real types

A real type has a set of values that is a subset of real numbers, which can be represented in floating-point notation with a fixed number of digits. A value's floating-point notation normally comprises three values—*M*, *B*, and *E*—such that $M \times B^E = N$, where *B* is always 2, and both *M* and *E* are integral values within the real

type's range. These *M* and *E* values further prescribe the real type's range and precision.

There are six kinds of real types: *Real*, *Single*, *Double*, *Extended*, *Comp*, and *Currency*. The real types differ in the range and precision of values they hold as shown in the following table:

Table 4.4 Real data types

Type	Range	Significant digits	Size in bytes
Real	2.9×10^{-39} .. 1.7×10^{38}	11–12	6
Single	1.5×10^{-45} .. 3.4×10^{38}	7–8	4
Double	5.0×10^{-324} .. 1.7×10^{308}	15–16	8
Extended	3.4×10^{-4932} .. 1.1×10^{4932}	19–20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19–20	8
Currency	-922337203685477.5808.. 922337203685477.5807	19–20	8

The *Comp* (computational) type is a 64-bit integer that holds integral values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (-2^{63} to $2^{63}-1$).

The *Currency* type is a fixed-point data type suitable for monetary calculations. It is stored as a scaled 64-bit integer with the four least-significant digits implicitly representing four decimal places. The range of *Currency* values is –922,337,203,685,477.5808 to 922,337,203,685,477.5807. When mixed with other real types in assignments and expressions, values of type *Currency* are automatically scaled by dividing or multiplying by 10000. Since numbers stored in the *Currency* format are exact representations, operations on *Currency* values are not subject to rounding errors.

Note The *Real* type is provided for backwards compatibility with earlier versions of Delphi and Borland Pascal. Since the storage format of the *Real* type is not native to the Intel CPU family, operations on *Real* type values are slower than the other floating-point types.

String types

Delphi supports three types of strings: *Short strings*, *long strings* and *wide strings*. Short strings, long strings and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions.

The short string type represents a statically-allocated string with maximum length between 1 and 255 characters, and a dynamic length between 0 and the maximum length. The short string type primarily exists for reasons of backward compatibility with earlier versions of Delphi and Borland Pascal.

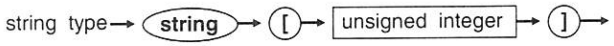
The long string type represents a dynamically-allocated string with a maximum length limited only by available memory. For new applications, it is recommended that you use the long string type.

Note A compiler directive, **$H**, controls whether the reserved word **string** represents a short string or a long string. In the default state, **{$H+}**, **string** represents a long string, and using the **string** keyword is the same as using the predefined identifier *AnsiString*. In the **{$H–}** state, **string** represents a short string with a maximum length of 255 characters, and using the **string** keyword is the same as using the predefined identifier *ShortString*.

The wide string type represents a string composed of Unicode characters. Unicode is a 16-bit character set system. ANSI characters, those which are used by short strings and long strings, are 8 bits wide. The 32-bit version of OLE uses Unicode for all strings. For ActiveX applications, strings must be of wide string type. As with long strings, wide strings are dynamically-allocated with a maximum length limited only by available memory.

Short string types

The declaration of a short string type specifies a maximum length between 1 and 255 characters. Variables of a short string type can contain strings with a dynamic length between 0 and the declared maximum length.



The predefined type *ShortString* denotes a short string type with a maximum length of 255 characters.

The number of bytes of storage occupied by a short string type variable is the maximum length of the short string type plus one.

When assigning a string value to a short string variable, the string value is truncated if it is longer than the declared maximum length of the short string variable.

You can index a short string variable with a single index expression, whose value must be in the range 0..*N*, where *N* is the declared maximum length of the short string. The type of a character accessed through indexing of a short string is *Char*. The index of the first character in a string is 1. The element at index 0 contains the dynamic length of the string, and for a short string, *Length(S)* is the same as *Ord(S[0])*. Assigning a value to the zeroth element of a short string alters the dynamic length of the string, but the compiler doesn't check whether the value is less than the declared maximum length of the string. It is possible to index a short string beyond its current dynamic length. The character values read in that case are undefined, and assignments to character positions beyond the current length do not affect the actual value of the short string variable.

The *Low* and *High* standard functions can be applied to a short string type variable. In this case, *Low* returns zero, and *High* returns the declared maximum length of the short string.

Long string types

The long string type is denoted by the reserved word **string** and by the predefined identifier *AnsiString*.

Note If you change the state of the **$H** compiler switch to {**$H-**}, the reserved word **string** denotes a short string with a maximum length of 255 characters. However, the predefined identifier *AnsiString* always denotes the long string type.

Long strings are dynamically allocated and have no declared maximum length. The theoretical maximum length of a long string value is 2GB (two gigabytes). In practice this means that the maximum length of a long string value is limited only by the amount of memory available to an application.

Management of the dynamically allocated memory associated with a long string variable is entirely automatic and requires no additional user code. The automatic management of long strings has the following characteristics:

- A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a non-empty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count.
- Long strings are reference counted. When a long string variable is assigned a new value, the reference count of its previous value (if non-empty) is decremented, and the reference count of its new value (if non-empty) is incremented. When the reference count of a string value reaches zero, the dynamically allocated memory that contains the string value is deallocated. Reference counting dramatically reduces string-data copying and memory consumption. The only data that is copied in a long string assignment is the 32-bit string value pointer, and any number of long string variables can reference the same value without consuming additional memory. For this reason, long string assignments typically execute faster than short string assignments.
- When indexing is used to change the value of a single character in a long string variable, a copy of the string value is first created if the string value's reference count is greater than one. This is known as *copy-on-write* semantics, and guarantees that the modification does not also modify other long string variables that reference the string value.
- Long string variables are always initialized to be empty when they are first created. This is true whether a long string variable is global, local, or part of a structure such as an array, record, or object.
- When long string variables go out of scope (such as upon exiting a function with local long string variables, or upon destroying an object that contains long string fields), the reference count of their values are automatically decremented. For a function this is true even if the function is exited because of an exception.

You can index a non-empty long string variable with a single index expression, whose value must be in the range 1..*N*, where *N* is the dynamic length of the long string. The type of a character accessed through indexing of a long string is *Char*. The index of the first character in a long string is 1. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the *Length* standard function, and to set the length of a long string you must use the *SetLength* standard procedure.

While it is possible to assign to a character position obtained by indexing a long string, it is not recommended to pass such a character as a **var** parameter.

When declaring long string fields in a record type, all such fields must reside in the non-variant part of the record type. In other words, long string fields or fields of a type that contains long strings are not allowed in a variant part of a record type.

Dynamic memory allocated for a long string value is automatically terminated by a null character (the null character is not part of the string, but rather is automatically stored right after the last character in the string). Because of the automatic null character termination, it is possible to directly typecast a long string value to a *PChar* value. The syntax of such a typecast is *PChar(S)*, where *S* is a long string expression. A *PChar* typecast returns a pointer to the first character of the long string value, and is guaranteed to return a pointer to a null-terminated string even if the string expression is empty.

The following example shows how *PChar* typecasts can be used to pass long string values to a function that expects null-terminated string parameters. *Caption* and *Message* are long string variables, and *MessageBox* is a Win32 API function defined in the *Windows* interface unit.

```
Caption := 'Hello world';
Message := 'This is a test of long strings';
MessageBox(0, PChar(Message), PChar(Caption), MB_OK);
```

It is also possible to typecast a long string to an untyped pointer, using the syntax *Pointer(S)*, where *S* is a long string expression. A *Pointer* typecast returns the address of the first character of the long string value. Unlike a *PChar* typecast, a *Pointer* typecast returns **nil** if the string expression is empty.

The lifetime of a pointer returned by a *PChar* or *Pointer* typecast depends on the argument of the typecast. If the argument is a long string *expression*, the pointer remains valid only within the statement in which the typecast is performed. This essentially limits such a typecast to use only in parameter expressions. If the argument is a long string *variable*, the pointer remains valid until a new value is assigned to the variable, or until the variable goes out of scope.

In general, the null-terminated string referenced by the pointer returned by a *PChar* or *Pointer* typecast should be considered *read-only*.

It is possible to use the pointer returned by a *PChar* or *Pointer* typecast to modify a corresponding long string, but it is safe to do so only in certain situations. Given a typecast of the form *PChar(S)* or *Pointer(S)*, the null-terminated string referenced by the result of the typecast can be modified only if all of the following requirements are satisfied:

- *S* is a long string *variable* (not an expression).
- The value of *S* is not empty. In other words, *S* must contain a string with a length greater than zero.
- The value of *S* is *unique*, that is the long string value has a reference count of one. Following a call to the *SetLength*, *SetString*, and *UniqueString* standard procedures, a long string variable's value is unique, and it is guaranteed to remain unique as long as the string variable is not referenced in a string expression.

- *S* has not been modified and has not gone out of scope since it was typecast.
- The characters modified all lie within the length of the string. In other words, when indexing the returned *PChar* value, index values must be between 0 and *Length(S)* – 1.

For further details on mixing long strings and null-terminated strings, see Chapter 17, "Using null-terminated strings."

Wide string types

The wide string type is denoted by the predefined identifier *WideString*. The wide string type represents a string composed of *Unicode* characters. Unicode is a 16-bit character set system. ANSI characters, those which are used by short strings and long strings, are 8 bits wide. For more information on the Unicode character set, see "Null-terminated wide character strings" on page 17-6.

Since the 32-bit version of OLE uses Unicode for all strings, strings must be of wide string type in any OLE automated properties and method parameters. Also, most OLE API functions use null-terminated wide strings.

As with long strings, wide strings are dynamically allocated with a maximum length limited only by available memory. However, wide strings are not reference counted — the dynamically allocated memory that contains the string is deallocated when the wide string goes out of scope.

In all other respects wide strings possess the same attributes as long strings. See "Long string types" on page 4-8.

Using strings

The ordering between any two string values is defined by the ordering relationship of the character values in corresponding positions. In two strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a higher or greater-than value. For example, 'BA' is greater than 'A'. Zero length strings are equal only to other zero length strings, and they hold the least string values.

Operators for the string types are described in "String operator" on page 6-6 and "Relational operators" on page 6-8.

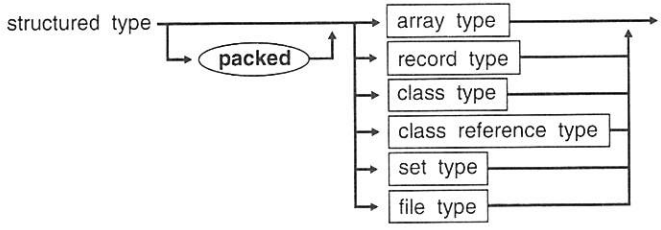
It is possible to mix short and long string types in assignments and expressions, but strings passed as **var** parameters must be of the appropriate type. In other words, it is not possible to pass a short string as a **var** parameter to a procedure or function that expects a long string, and vice versa.

A short string can be explicitly converted to a long string using a typecast of the form *AnsiString(S)*, where *S* is a short string expression. Also, in the default **{$H+}** state, a typecast of the form ***string(S)***, will convert a short string expression to a long string.

A long string can be explicitly converted to a short string using a typecast of the form *ShortString(S)*, where *S* is a long string expression. Also, in the **{$H–}** state, a typecast of the form ***string(S)***, will convert a long string expression to a short string.

Structured types

A structured type, characterized by its structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring.

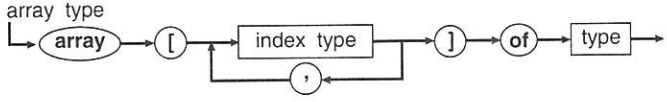


The word **packed** in a structured type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a variable of this type. By default, Object Pascal aligns components of structured types on word or double-word boundaries for faster access. Adding **packed** to a structured type's declaration overrides such alignment for that type.

Class types and class reference types are the cornerstones of object-oriented programming in Object Pascal. They are described in full in Chapter 10, "Class types."

Array types

Arrays have a fixed number of components of one type—the *component type*. In the following syntax diagram, the component type follows the word **of**.



The index types, one for each dimension of the array, specify the number of elements. Valid index types are all ordinal types. The array can be indexed in each dimension by all values of the corresponding index type; therefore, the number of elements is the product of the number of values in each index type.

The following is an example of an array type:

```
array[1..100] of Real
```

If an array type's component type is also an array, you can treat the result as an array of arrays or as a single multidimensional array. For example,

```
array[Boolean] of array[1..10] of array[Size] of Real
```

is interpreted the same way by the compiler as

```
array[Boolean,1..10,Size] of Real
```

You can also express

```
packed array[1..10] of packed array[1..8] of Boolean
```

as

```
packed array[1..10,1..8] of Boolean
```

You access an array's components by supplying the array's identifier with one or more indexes in brackets. See "Arrays, strings, and indexes" on page 5-3.

When applied to an array-type identifier or a variable reference of an array type, the *Low* and *High* standard functions return the low and high bounds of the index type of the array.

An array type of the form

```
packed array[M..N] of Char
```

where *M* is less than *N* is called a *packed string type*. A packed string type has certain properties not shared by other array types, as explained in "Identical and compatible types" on page 4-31.

An array type of the form

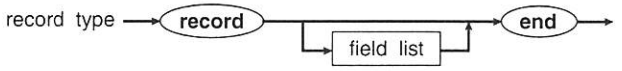
```
array[0..X] of Char
```

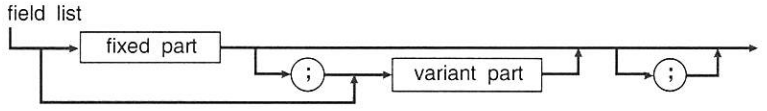
where X is a positive nonzero integer is called a *zero-based character array*. Zero-based character arrays are used to store *null-terminated strings*. A zero-based character array is compatible with a *PChar* value. For a complete discussion of this topic, read Chapter 17, "Using null-terminated strings."

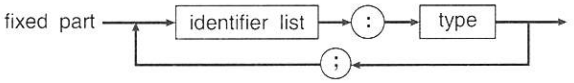
A parameter declared using the **array of** *T* syntax is an *open-array parameter*. Open-array parameters allow arrays of varying sizes to be passed to the same procedure or function. Read about open-array parameters on page 9-9.

Record types

A record type comprises a set number of components, or fields, that can be of different types. The record-type declaration specifies the type of each field and the identifier that names the field.





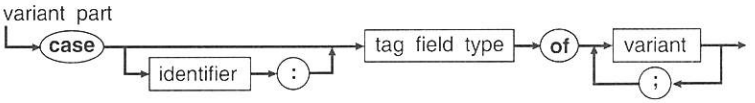


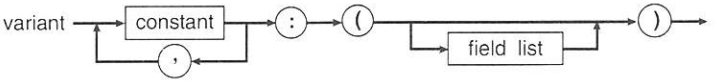
The fixed part of a record type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

The following is an example of a record type:

```
type
  TDateRec = record
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
  end;
```

The variant part shown in the syntax diagram of a record-type declaration distributes memory space for more than one list of fields, so the information can be accessed in more ways than one. Each list of fields is a *variant*. The variants overlay the same space in memory, and all fields of all variants can be accessed at all times.





You can see from the diagram that each variant is identified by at least one constant. All constants must be distinct and of an ordinal type compatible with the tag field type. Variant and fixed fields are accessed the same way.

An optional identifier, the *tag field identifier*, can be placed in the variant part. If a tag field identifier is present, it becomes the identifier of an additional fixed field—the tag field—of the record. The program can use the tag field's value to show which variant is active at a given time. Without a tag field, the program selects a variant by another criterion.

Note Fields in the variant part of a record cannot be of a long string type or of the type *Variant*. Likewise, fields in the variant part of a record cannot be of a structured type that contains long string or *Variant* components.

Some record types with variants follow:

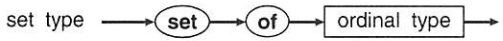
```
type
  TPerson = record
```

```
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
      True: (BirthPlace: string[40]);
      False: (Country: string[20];
        EntryPort: string[20];
        EntryDate: TDate;
        ExitDate: TDate);
  end;

  TPolygon = record
    X, Y: Real;
    case Kind: Figure of
      TRectangle: (Height, Width: Real);
      TTriangle: (Side1, Side2, Angle: Real);
      TCircle: (Radius: Real);
  end;
```

Set types

A set type's range of values is the power set of a particular ordinal type (the base type). The power set is the set of all possible subsets of values of the base type including the empty set. Therefore, each possible value of a set type is a subset of the possible values of the base type.

A variable of a set type can hold from none to all the values of the set. Set-type operators are described in "Set operators" on page 6-7. "Set constructors" on page 6-11 shows how to construct set values.

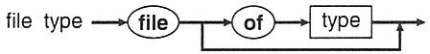


The base type must not have more than 256 possible values, and the ordinal values of the upper and lower bounds of the base type must be within the range 0 to 255.

Every set type can hold the value [ ], which is called the empty set.

File types

A file type consists of a linear sequence of components of the component type, which can be of any type except a file type, any structured type with a file-type component, or an object type. The number of components isn't set by the file-type declaration.

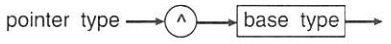


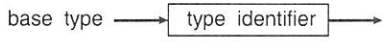
If the word **of** and the component type are omitted, the type denotes an untyped file. Untyped files are low-level I/O (input/output) channels primarily used for direct access to any disk file regardless of its internal format.

The standard file type *Text* (or *TextFile*) signifies a file containing characters organized into lines. Text files use special I/O procedures, which are discussed in Chapter 16, "Input and output."

Pointer types

A pointer type defines a set of values that point to dynamic variables of a specified type called the *base type*. A pointer-type variable contains the memory address of a dynamic variable.





If the base type is an undeclared identifier, it must be declared in the same type declaration part as the pointer type.

You can assign a value to a pointer variable with the *New* procedure, the @ operator, the *Ptr* function, or the *GetMem* procedure. *New* allocates a new memory area in the application heap for a dynamic variable and stores the address of that area in the pointer variable. The @ operator directs the pointer variable to the memory area containing any existing variable or procedure or function entry point, including variables that already have identifiers. *Ptr* points the pointer variable to a specific memory address. *GetMem* creates a new dynamic variable of a specified size, and puts the address of the block in the pointer variable.

The reserved word **nil** denotes a pointer-valued constant that doesn't point to anything.

Type Pointer

The predefined type *Pointer* denotes an untyped pointer; that is, a pointer that doesn't point to any specific type. Variables of type *Pointer* can't be dereferenced; writing the pointer symbol ^ after such a variable is an error. Generic pointers, however, can be typecast to allow dereferencing. Like the value denoted by the word **nil**, values of type *Pointer* are compatible with all other pointer types. For the syntax of referencing the dynamic variable pointed to by a pointer variable, see "Pointers and dynamic variables" on page 5-5.

Character pointers

Object Pascal defines two fundamental character-pointer types and a generic character-pointer type. The difference between fundamental and generic types is explained on page 4-1.

Character-pointer types are simply pointers to character types, but Object Pascal supports a set of *extended syntax* rules to facilitate handling of null-terminated strings using character-pointer types. For a complete discussion of this topic, see Chapter 17, "Using null-terminated strings."

The fundamental character-pointer types are *PAnsiChar* and *PWideChar*. *PAnsiChar* is a pointer to a null-terminated string of characters of type *AnsiChar*. *PWideChar* is a pointer to a null-terminated string of characters of type *PWideChar*.

The generic character-pointer type is *PChar*, which represents a pointer to a null-terminated string of characters of type *Char*. The *System* unit declares the pointer-character types as

```
type
  PAnsiChar = ^AnsiChar;
  PWideChar = ^WideChar;
  PChar = PAnsiChar;
```

That is, the fundamental character-pointer types are pointers to the fundamental character types (or to null-terminated strings of such characters), and the generic character-pointer type is a pointer to the generic character type.

In this version of Object Pascal, *PChar* is equivalent to *PAnsiChar*.

Standard pointers

When you define a structure or a data type in Pascal, you should also define a pointer to that data type. Many advanced programming techniques, such as linked lists of dynamically allocated records, may require you use the pointer instead of the variable itself.

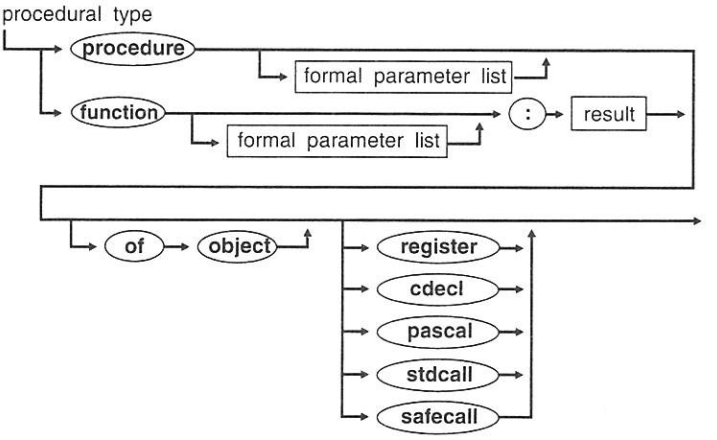
When you define a variable the compiler generates the necessary code to initialize and finalize these special types automatically. When you write code that allocates and frees the pointers to dynamic structures like *AnsiString*, and *Variant*, they require special initialization and finalization. See also online help for the *Initialize* procedure.

Table 4.5 Some pointers declared in Delphi

Pointer	Points to
PAnsiString	Points to an *AnsiString* variable.
PByteArray	Points to a variable of type *TByteArray*. Often used to typecast dynamically allocated blocks of memory for array access.
PCurrency	Points to a variable of type *Currency*.
PExtended	Points to a variable of type *Extended*.
PShortString	Points to a variable of type *ShortString*. Mainly used when porting Delphi 1.0 16-bit code that uses *PString* to Delphi 2.0.
PTextBuf	Points to a variable of type *TextBuf*. *TextBuf* is the internal buffer used in the *TTextRec* text file record.
PVarRec	Points to a variable of type *TVarRec*.
PVariant	Points to a variable of type *Variant*.
PWordArray	Points to a variable of type *TWordArray*. Often used to typecast dynamically allocated memory blocks for use as an array of word-sized (2 byte unsigned) values.

Procedural types

Object Pascal allows procedures and functions to be treated as entities that can be assigned to variables and passed as parameters. Such actions are made possible through *procedural types*.



The syntax for a procedural-type declaration is the same as a that of a procedure or function header, except that the identifier after the **procedure** or **function** keyword is omitted. A procedural-type declaration can optionally specify a calling convention. The default calling convention is **register**.

There are two categories of procedural types: *Global procedure pointers* and *method pointers*.

Global procedure pointers

A procedural type declared without the **of object** clause is called a global procedure pointer. A global procedure pointer can reference a global procedure or function, and is encoded as a pointer that stores the address of a global procedure or function. Some examples of global procedure pointer types follow:

```
type
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
```

Method pointers

A procedural type declared with the **of object** clause is called a method pointer. A method pointer can reference a procedure or function method of an object, and is encoded as two pointers. The first pointer stores the address of a method, and the second pointer stores a reference to the object that the method belongs to. Some examples of method pointer types follow:

```
type
  TMethod = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

Procedural values

A variable of a procedural type can be assigned a *procedural value*. Procedural values can be one of the following:

- The value **nil**
- A variable reference of a procedural type
- A global procedure or function identifier
- A method designator

In the context of procedural values, a global procedure or function identifier denotes a global procedure pointer value, and a method designator denotes a method pointer value. For example, given the following declarations:

```
type
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ...
  end;

var
  MainForm: TMainForm;
  MathFunc: TMathFunc;
  OnClick: TNotifyEvent;

function Tan(Angle: Double): Double;
begin
  Result := Sin(Angle) / Cos(Angle);
end;
```

The variables *MathFunc* and *OnClick* can be assigned values as follows:

```
MathFunc := Tan;
OnClick := MainForm.ButtonClick;
```

and calls can be made using *MathFunc* and *OnClick* as follows:

```
X := MathFunc(X);        { Equivalent to X := Tan(X) }
OnClick(Self);           { Equivalent to MainForm.ButtonClick(Self) }
```

Using a procedural variable that contains the value **nil** in a procedure statement or function call results in an error. The value **nil** is intended to indicate that a procedural variable is unassigned, and whenever there is a possibility that a procedural value is **nil**, procedure statements or function calls involving that procedural variable should be guarded by a test:

```
if Assigned(OnClick) then OnClick(Self);
```

The *Assigned* standard function returns *True* if the given procedural variable has been assigned a procedural value, or *False* if the procedural variable contains **nil**.

Procedural type compatibility

To be considered compatible, procedural types must have the same calling convention, the same number of parameters, and parameters in corresponding positions must be of identical types. Finally, the result types of functions must be identical. Parameter names have no significance when determining procedural-type compatibility.

The value **nil** is compatible with any procedural type.

Global procedure pointer types and method pointer types are always mutually incompatible. In other words, a global procedure or function cannot be assigned to a method pointer variable, and a method cannot be assigned to a global procedure pointer variable.

Standard procedures and functions and nested procedures and functions cannot be used as procedural values.

Standard procedures and functions are the ones declared by the *System* unit, such as *WriteLn*, *ReadLn*, *Chr*, and *Ord*. To use a standard procedure or function as a procedural value, write a "shell" around it. For example, the following function *FSin* is assignment-compatible with the *TMathFunc* type declared above.

```
function FSin(X: Double): Double;
begin
  FSin := Sin(X);
end;
```

A procedure or function is *nested* when it's declared within another procedure or function. Such nested procedures and functions can't be used as procedural values.

Variant types

The *Variant* type is capable of representing values that change type *dynamically*. Whereas a variable of any other type is *statically* bound to that type, a variable of the *Variant* type can assume values of differing types at runtime. The *Variant* type is most commonly used in situations where the actual type to be operated upon varies or is unknown at compile-time.

The predefined identifier *Variant* is used to denote the variant type. Variants have the following characteristics:

- Variants can contain integer values, real values, string values, Boolean values, date-and-time values, and OLE Automation objects. In addition, variants can contain arrays of varying size and dimension with elements of any of these types.
- The special variant value *Unassigned* is used to indicate that a variant has not yet been assigned a value, and the special variant value *Null* is used to indicate unknown or missing data.
- Variants can be combined with other variants and with integer, real, string, and Boolean values in expressions, and the compiler will automatically generate code that performs the necessary type conversions.

- When a variant contains an OLE Automation object, the variant can be used to get and set properties of the object, and to invoke methods on the object.
- Variant variables are *always* initialized to be *Unassigned* when they are first created. This is true whether a variant variable is global, local, or part of a structure such as an array, record, or object.

The following section of code demonstrates the use of variants and some of the automatic type conversions that are performed when variants are combined with other types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1;{ Integer value }
  V2 := 1234.5678;{ Real value }
  V3 := 'Hello world';{ String value }
  V4 := '1000';{ String value }
  V5 := V1 + V2 + V4;{ Real value 2235.5678 }
  I := V1;{ I = 1 }
  D := V2;{ D = 1234.5678 }
  S := V3;{ S = 'Hello world' }
  I := V4;{ I = 1000 }
  S := V5;{ S = '2235.5678' }
end;
```

Note that while variants offer great flexibility, they also consume more memory than regular variables, and operations on variants are substantially slower than operations on statically typed values.

Values in variants

A variable of type *Variant* occupies 16 bytes of memory, and its internal representation consists of a type code and a value (or a reference to a value) of the type given by the type code. The standard function *VarType* returns a variant's type code. Table 4.6 lists the variant type constants and values, and the meaning of each type code.

Table 4.6 Variant type constants and values

VarType	Value	Contents of variant
varEmpty	$0000	The variant is *Unassigned*.
varNull	$0001	The variant is *Null*.
varSmallint	$0002	16-bit signed integer (type *Smallint*).
varInteger	$0003	32-bit signed integer (type *Integer*).
varSingle	$0004	Single-precision floating-point value (type *Single*).
varDouble	$0005	Double-precision floating-point value (type *Double*).
varCurrency	$0006	Currency floating-point value (type *Currency*).
varDate	$0007	Date and time value (type *TDateTime*).

Table 4.6 Variant type constants and values (continued)

VarType	Value	Contents of variant
varOleStr	$0008	Reference to an OLE string (a dynamically allocated Unicode string).
varDispatch	$0009	Reference to an OLE automation object (an IDispatch interface pointer).
varError	$000A	Operating system error code.
varBoolean	$000B	16-bit Boolean (type *WordBool*).
varVariant	$000C	Variant (used only with variant arrays).
varUnknown	$000D	Reference to an unknown OLE object (an IUnknown interface pointer).
varByte	$0011	8-bit unsigned integer (type *Byte*).
varString	$0100	Reference to a dynamically allocated long string (type *AnsiString*).
varTypeMask	$0FFF	Bit mask for extracting type code.
varArray	$2000	Bit indicating variant array.

The *varXXXX* constants returned by the *VarType* standard function are defined in the *System* unit. Note that future versions of Delphi may define additional type codes, so be careful not to write code that depends on only these codes being returned.

The *varArray* bit position is set if the variant contains an array of the given type. The *varTypeMask* bit mask is used to extract the actual type code from a value returned by the *VarType* function. For example, the following expression is *True* if *V* contains a *Double* or an array of *Double*:

```
if VarType(V) and varTypeMask = varDouble then ...
```

The *TVarData* record defined in the *System* unit can be used to typecast a *Variant* variable to gain access to its internal representation. For further details, see the description of *TVarData* in online help.

Variant type conversions

All integer, real, string, character, and Boolean types are assignment compatible with the *Variant* type. Table 4.7 lists the types that can be assigned to a *Variant*, and the resulting variant type codes.

Table 4.7 Expression types and resulting variant type codes

Expression type	Variant type code
integer types	varInteger
real types except Currency	varDouble
Currency type	varCurrency
string and character types	varString
Boolean types	varBoolean

An expression can be explicitly cast to type *Variant* using a typecast of the form *Variant(X)*, where *X* is an expression of one of the types listed in Table 4.7.

A *Variant* is assignment compatible with all integer, real, string, and Boolean types. Table 4.8 through Table 4.12 show the type-conversion rules that govern conversions of *Variant* values to other types.

Table 4.8 Converting a variant to an integer type value

Variant type	Result
varEmpty	0
varNull	Raises an *EVariantError* exception.
varByte varSmallint varInteger varError	Converts one integer format to another, raises an *EVariantError* exception if value does not fit in destination format.
varSingle varDouble varCurrency	Rounds real value to nearest integer, raises an *EVariantError* exception if result does not fit in destination format.
varDate	Interprets date and time value as a *Double*, rounds value to nearest integer, raises an *EVariantError* exception if result does not fit in destination format.
varOleStr varString	Converts string to integer, raises an *EVariantError* exception if string is not a valid integer value or if result does not fit in destination format.
varBoolean	0 for *False*, –1 for *True* (255 if destination type is *Byte*).

Table 4.9 Converting a variant to a real type value

Variant type	Result
varEmpty	0.
varNull	Raises an *EVariantError* exception.
varByte varSmallint varInteger varError	Converts integer to real.
varSingle varDouble varCurrency	Converts from one real format to another, raises an *EVariantError* exception if value does not fit in destination format.
varDate	Interprets value as a *Double*, converts value to destination format, raises an *EVariantError* exception if value does not fit in destination format.
varOleStr varString	Converts string to real using the Regional Settings in the Windows Control Panel, raises an *EVariantError* exception if string is not a valid real value or if result does not fit in destination format.
varBoolean	0 for False, –1 for True.

Table 4.10 Converting a variant to a date and time value

Variant type	Result
varEmpty	12/30/1899 12:00:00 am.
varNull	Raises exception.
varByte varSmallint varInteger varError	Converts integer to *Double*, interprets result as a date and time value.

Table 4.10 Converting a variant to a date and time value (continued)

Variant type	Result
varSingle varDouble varCurrency	Converts value to *Double*, interprets result as a date and time value.
varDate	No translation.
varOleStr varString	Converts string to a date and time value using the Regional Settings in the Windows Control Panel.
varBoolean	Converts Boolean to *Double*, interprets result as a date and time value.

Table 4.11 Converting a variant to a string type value

Variant type	Result
varEmpty	Empty string.
varNull	Raises exception.
varByte varSmallint varInteger varError	Converts integer value to its string representation.
varSingle varDouble varCurrency	Converts real value to its string representation using the Regional Settings in the Windows Control Panel.
varDate	Converts date and time value to its string representation using the Regional Settings in the Windows Control Panel.
varOleStr	Converts from Unicode to ANSI if destination type is *varString*.
varString	Converts from ANSI to Unicode if destination type is *varOleStr*.
varBoolean	'0' for *True*, '–1' for *False*.

Table 4.12 Converting a variant to a Boolean type value

Variant type	Result
varEmpty	*False*.
varNull	Raises exception.
varByte varSmallint varInteger varError	*False* if value is zero, *True* if value is non-zero.
varSingle varDouble varCurrency	*False* if value is zero, *True* if value is non-zero.
varDate	Interprets value as a *Double*, returns *False* if value is zero, *True* if value is non-zero.
varOleStr varString	*False* if string contains 'false' (case insensitive) or a numeric string that evaluates to zero, *True* if string contains 'true' (case insensitive) or a numeric string that evaluates to non-zero, otherwise raises an *EVariantError* exception.
varBoolean	No translation.

A *Variant* value can be explicitly cast to an integer, real, string, or Boolean type using a typecast of the form *TypeName(V)*, where *TypeName* is an integer, real, string, or Boolean type identifier and *V* is an expression of type *Variant*. Furthermore, the *VarAsType* standard function and the *VarCast* standard procedure can be used to change the internal representation of a variant. Table 4.8 through Table 4.12 list the rules that govern all such type conversions.

If a variant contains a reference to an OLE Automation object (the *varDispatch* type code), any attempt to convert the variant to another type will first fetch the value of the object's *default property* and then convert that value to the requested type. If the given OLE Automation object has no default property, an *EVariantError* exception is raised.

Variant expressions

Variants can participate in expressions. The following operators support operands of type *Variant*:

+ - * / div mod shl shr and or xor not = <> < > <= >=

For operators that take two operands, if one operand is of type *Variant*, the other operand is automatically converted to type *Variant* using the rules set forth in Table 4.7. The result type of a non-relational operation (all operators shown above but the last six) on *Variant* values is always *Variant*. The result type of a relational operation on *Variant* values is always *Boolean*.

For all non-relational operators, if one or both operands are *Unassigned*, an *EVariantError* exception is raised. In other words, no operations other than comparisons are allowed on *Unassigned* variants.

Also for all non-relational operators, if one or both operands are *Null*, the result of the operation is *Null*. In other words, *Null* values propagate through expressions, and the presence of a *Null* value in an expression causes the entire expression to become *Null*.

When performing a double operand operation, the *common* type of the two *Variant* operands governs the operation. The common type is determined using the matrix shown in Table 4.13. When reading Table 4.13, variant type codes *varSmallint*, *varInteger*, and *varByte* map to *Integer*, *varSingle* and *varDouble* map to *Double*, and *varOleStr* and *varString* map to *String*.

Table 4.13 Variant operation type matrix

	Integer	Double	Currency	String	Boolean	Date
Integer	Integer	Double	Currency	Double	Integer	Date
Double	Double	Double	Currency	Double	Double	Date
Currency	Currency	Currency	Currency	Currency	Currency	Date
String	Double	Double	Currency	String	Boolean	Date
Boolean	Integer	Double	Currency	Boolean	Boolean	Date
Date	Date	Date	Date	Date	Date	Date

For example, in the operation *V1* + *V2*, if the type code of *V1* is *varInteger* and the type code of *V2* is *varString*, the common type used to perform the operation is *Double*.

For non-relational operators, once the common type is established the operation proceeds as described in Table 4.14.

Table 4.14 Non-relational variant operator results

Common type	Operator results
Integer	For all operators except /, the operands are converted to *Integer*, and the result type is *Integer*, or *Double* if the result does not fit in a 32-bit signed integer. For the / operator, the operation is performed as a *Double* operation.
Double	For the +, –, *, and / operators, the operands are converted to *Double* and the result type is *Double*. For all other operators, the operation is performed as an *Integer* operation.
Currency	For the +, –, *, and / operators, the operands are converted to *Currency* and the result type is *Currency*, or *Double* when dividing two *Currency* values. For all other operators, the operation is performed as an *Integer* operation.
String	For the + operator, if both operands are strings, the result is the concatenation of the two strings. Otherwise, and for all other operators, the operation is performed as a *Double* operation.
Boolean	For the **and**, **or**, and **xor** operators, the operands are converted to *Boolean* and the result type is *Boolean*. For all other operators, the operation is performed as a *Double* operation.
Date	For the + and – operators, the operands are converted to *Date* and the result type is *Date*, or *Double* when subtracting two *Date* values. For all other operators, the operation is performed as a *Double* operation.

For relational operators, both variants are converted to the common type, and the resulting values are compared to produce a *Boolean* result. The *Unassigned* value compares less than all other values. The *Null* value compares greater than *Unassigned* and less than all other values.

For the unary minus (–) operator, strings are converted to *Double* before the operation, and Booleans are converted to *Integer* before the operation.

For the **not** operator, if the type code of the variant is *varBoolean*, a logical negation operation is performed. For all other type codes, the variant is converted to *Integer* and a bitwise negation operation is performed.

OleVariant

An *OleVariant* type is a variant that contains only OLE variant compatible types. When a *Variant* is assigned to an *OleVariant*, all non-OLE variant compatible types are converted to their OLE variant compatible counterparts. For example, An *OleVariant* type cannot contain an *AnsiString* type. When a variant that contains an *AnsiString* is assigned to an *OleVariant*, the *AnsiString* is converted to a *WideString*.

Variant arrays

Variants can contain arrays of varying size and dimension with elements of any of the variant base types. The elements of a variant array are all of the same type, but if

the element type is *Variant*, individual elements can of course contain different kinds of data (including other variant arrays). See the *Delphi Library Reference* for the variant array standard procedures and functions, which are all defined in the *System* unit.

Variant arrays are typically created using the *VarArrayCreate* standard procedure, for example:

```
var
  A, B: Variant;
  I: Integer;
begin
  A := VarArrayCreate([0, 9], varInteger);
  for I := 0 to 9 do A[I] := I * I;
  B := VarArrayCreate([1, 3, 0, 9], varVariant);
  for I := 0 to 9 do B[1, I] := I;
  for I := 0 to 9 do B[2, I] := Sqrt(I);
  for I := 0 to 9 do B[3, I] := Format('Value=%d', [I]);
  ...
end;
```

Note The element type of a variant array cannot be *varString*. To create variant arrays of strings, you must use the *varOleStr* type code.

When a variant contains an array, elements of the array can be accessed by following the variant with one or more index expressions, enclosed in square brackets and separated by commas. Index expressions are always of type *Integer*. When indexing a variant, an *EVariantError* exception is raised if the variant does not contain a variant array, if an incorrect number of index expressions are specified, or if one or more of the index expressions are not within the bounds of the corresponding dimension.

Variant array elements can be accessed in expressions and assigned new values using assignment statements. Note however, that it is not possible to pass a variant array element as a **var** parameter.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, a copy of the entire array is made, possibly consuming a lot of memory. For this reason, whenever possible you should avoid assigning variant arrays to other variants, and unless there is a good reason not to, you should pass variant arrays as **var** or **const** parameters.

A variant array can be resized using the *VarArrayRedim* standard procedure. *VarArrayRedim* allows you to change the high bound of the rightmost (last) dimension of a variant array. The bounds of other dimensions cannot be changed. Existing elements of an array are preserved across a resize operation. The code fragment below demonstrates the use of *VarArrayRedim*.

```
var
  A: Variant;
  I: Integer;
begin
  A := VarArrayCreate([0, 4], varOleStr);
  for I := 0 to 4 do A[I] := 'Initial';
  ...
  VarArrayRedim(A, 9);
  for I := 5 to 9 do A[I] := 'Additional';
```

```
  ...
end;
```

The *VarArrayDimCount*, *VarArrayLowBound*, and *VarArrayHighBound* standard functions allow you to examine the number of dimensions in a variant array, and the bounds of each dimension. This may be useful when writing general-purpose variant array manipulation routines, such as the *VarArraySum* function shown below.

```
function VarArraySum(const A: Variant): Double;
var
  I: Integer;
begin
  if VarArrayDimCount(A) <> 1 then
    raise Exception.Create('One-dimensional variant array expected');
  Result := 0;
  for I := VarArrayLowBound(A, 1) to VarArrayHighBound(A, 1) do
    Result := Result + Double(A[I]);
end;
```

Using the *VarArrayLock* standard function and the *VarArrayUnlock* standard procedure you can gain direct access to the data in a variant array. The *VarArrayLoadFile* function shown below loads the contents of a file into a variant array of bytes. It uses *VarArrayLock* and *VarArrayUnlock* to read the file directly into the array.

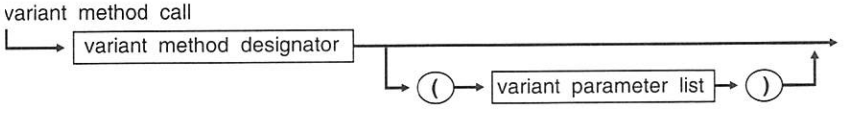
```
function VarArrayLoadFile(const FileName: string): Variant;
var
  F: file;
  Size: Integer;
  Data: PChar;
begin
  AssignFile(F, FileName);
  Reset(F, 1);
  try
    Size := FileSize(F);
    Result := VarArrayCreate([0, Size - 1], varByte);
    Data := VarArrayLock(Result);
    try
      BlockRead(F, Data^, Size);
    finally
      VarArrayUnlock(Result);
    end;
  finally
    CloseFile(F);
  end;
end;
```

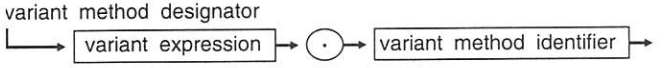
Variant arrays with an element type of *varByte* (like the one created by the function above) are the preferred method of passing binary data between OLE Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the *VarArrayLock* and *VarArrayUnlock* routines.

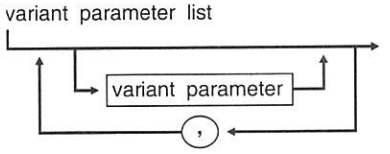
Variants and OLE Automation objects

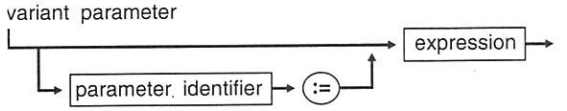
When a variant contains a reference to an OLE Automation object, Delphi allows you to call methods and get and set properties of the object. You enable this by using the *OleAuto* unit, that is by including a reference to *OleAuto* in the **uses** clause of one of your units or in the **uses** clause of your program or library.

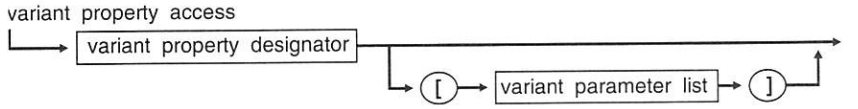
The syntax of an OLE Automation object method call or property access is analogous to that of a normal method call or property access.

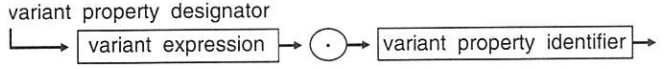












An example of a section of code that uses OLE Automation method calls follows below. Notice the use of the *CreateOleObject* function (defined in the *OleAuto* unit) to create a variant that contains a reference to an OLE Automation object.

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
```

```
    Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

An OLE Automation object method call is *late bound*, and requires no previous declaration of the method and its parameters. Contrary to a normal method call, in an OLE Automation method call the compiler allows *any* method identifier and *any* number and type of parameters to be specified. Whether the method call will actually succeed is not determined until it is executed at runtime.

Identifiers for OLE Automation object methods, properties, and named parameters are allowed to contain alphabetical characters from an international character set such as á, ü, and Ø.

OLE Automation method calls support two types of parameters: *Positional* parameters and *named* parameters. A positional parameter is simply an expression. A named parameter consists of a parameter identifier followed by a colon-equals symbol (:=) followed by an expression. Positional parameters must precede named parameters in an OLE Automation method call. It is possible to omit positional parameters by leaving parameter positions empty and by writing fewer parameter expressions than are expected by the method call. Some examples of positional and named parameters follow below.

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

The first call supplies one positional parameter. The second call supplies two positional parameters. The third call supplies four positional parameters, whereof the middle two are omitted. The fourth call supplies one positional parameter and one named parameter. Finally, the fifth call supplies no positional parameters and two named parameters. Note that using named parameters it is possible to write the parameters in any order.

Note OLE Automation servers often do not support named parameters. Likewise, some OLE Automation servers do not allow you to omit positional parameters.

OLE Automation method call parameters can be of the following types: Integer, real, string, Boolean, and variant. A parameter is passed *by reference* if the parameter expression consists only of a variable reference, and if the variable reference is of type *Byte*, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *AnsiString* (or **string**), *WordBool*, or *Variant*. If a parameter expression is not just a variable reference, or if the expression is not of one of the above types, the parameter is passed *by value*. When a parameter is passed by reference, the corresponding variable can be modified by the method to which a call is made.

Note Passing a parameter by reference to a method that expects a value parameter simply causes OLE to fetch the value from the reference parameter. The reverse situation, however, causes an error. In other words, it is an error to pass a value parameter where a reference parameter is expected.

Access to properties of an OLE Automation object follows the same rules as method calls. When a property access is used in an expression, the value of the property is

read, and when a property access is used on the left hand side in an assignment statement, the value of the property is *written*. For array properties, the index parameters list must be enclosed in square brackets.

As explained previously, the Delphi compiler allows any method or property identifier and any number and type of parameters to be specified in an OLE Automation method call or property access. The call information is simply packaged up by the compiler, and not until the call or property access is executed at runtime is it known whether it will succeed. If an OLE Automation method call or property access fails, an *EOleError* exception is raised. A call or property access may fail for any of the reasons listed below.

- The variant expression specified in the variant method designator or variant property designator does not currently reference an OLE Automation object.
- The method or property identifier is not supported by the OLE Automation object.
- An incorrect number of parameters were specified, or the type(s) of one or more parameters were incorrect.
- One or more required positional parameters were omitted.
- Named parameters were specified, but are not supported by the OLE Automation object.
- A method call was used in an expression, but the method did not return a value.
- The method or property access was successfully called, but returned an exception.

When an OLE Automation method call or property access fails, the *EOleError* exception object contains an error message that explains the reason for the failure.

Identical and compatible types

Two types can be the same, and this sameness (identity) is mandatory in some contexts. At other times, the two types need only be compatible or merely assignment-compatible. They are identical when they are declared with, or their definitions stem from, the same type identifier.

Type identity

Type identity is required only between actual and formal variable parameters in procedure and function calls.

Two types—say, *T1* and *T2*—are identical if one of the following is true: *T1* and *T2* are the same type identifier; *T1* is declared to be equivalent to a type identical to *T2*.

The second condition connotes that *T1* doesn't have to be declared directly to be equivalent to *T2*. The type declarations

```
T1 = Integer;
T2 = T1;
T3 = Integer;
```

```
T4 = T2;
```

result in *T1*, *T2*, *T3*, *T4*, and *Integer* as identical types. The type declarations

```
T5 = set of Char;
T6 = set of Char;
```

don't make *T5* and *T6* identical because **set of** Char isn't a type identifier. Two variables declared in the same declaration, for example,

```
V1, V2: set of Char;
```

are of identical types—unless the declarations are separate. The declarations

```
V1: set of Char;
V2: set of Char;
V3: Integer;
V4: Integer;
```

mean *V3* and *V4* are of identical type, but not *V1* and *V2*.

Type compatibility

Compatibility between two types is sometimes required, such as in expressions or in relational operations. Type compatibility is important, however, as a precondition of assignment compatibility.

Type compatibility exists when at least one of the following conditions is true:

- Both types are the same.
- Both types are real types.
- Both types are integer types.
- One type is a subrange of the other.
- Both types are subranges of the same host type.
- Both types are set types with compatible base types.
- Both types are packed string types with an identical number of components.
- One type is a string type and the other is either a string type, packed string type, or *Char* type.
- One type is *Pointer* and the other is any pointer type.
- Both types are class types or class reference types, and one type is derived from the other.
- One type is *PChar* and the other is a zero-based character array of the form **array**[0..X] **of** *Char*.
- Both types are pointers to identical types. (This applies only when type-checked pointers are enabled with the {**$T+**} directive.)
- Both types are procedural types with identical result types, an identical number of parameters, and a one-to-one identity between parameter types.

- One type is *Variant* and the other is either an integer type, real type, string type, or Boolean type.

Assignment compatibility

Assignment compatibility is necessary when a value is assigned to something, such as in an assignment statement or in passing value parameters.

A value of type T_2 is assignment-compatible with a type T_1 (that is, $T_1 := T_2$ is allowed) if any of the following are true:

- T_1 and T_2 are identical types and neither is a file type or a structured type that contains a file-type component at any level of structuring.
- T_1 and T_2 are compatible ordinal types, and the values of type T_2 falls within the range of possible values of T_1 .
- T_1 and T_2 are real types, and the value of type T_2 falls within the range of possible values of T_1.
- T_1 is a real type, and T_2 is an integer type.
- T_1 and T_2 are string types.
- T_1 is a string type, and T_2 is a *Char* type.
- T_1 is a string type, and T_2 is a packed string type.
- *T1* is a long string type, and *T2* is of type *PChar*.
- T_1 and T_2 are compatible, packed string types.
- T_1 and T_2 are compatible set types, and all the members of the value of type T_2 fall within the range of possible values of T_1 .
- T_1 and T_2 are compatible pointer types.
- T_1 is a class type and T_2 is a class type derived from T_1.
- T_1 is a class reference type and T_2 is a class reference type derived from T_1.
- T_1 is a *PChar* and T_2 is a string constant.
- T_1 is a *PChar* and T_2 is a zero-based character array of the form **array**[0..X] **of** *Char*.
- T_1 and T_2 are compatible procedural types.
- T_1 is a procedural type, and T_2 is a procedure or function with an identical result type, an identical number of parameters, and a one-to-one identity between parameter types.
- *T1* is *Variant*, and *T2* is an integer type, real type, string type, or Boolean type.
- *T1* is an integer type, real type, string type, or Boolean type, and *T2* is *Variant*.

A compile-time error occurs when assignment compatibility is necessary and none of the items in the preceding list are true.

The type declaration part

Programs, procedures, functions, and methods that declare types have a *type declaration part*. This is an example of a type declaration part:

```
type
  TRange = Integer;
  TNumber = Integer;
  TColor = (Red, Green, Blue);
  TCharVal = Ord('A')..Ord('Z');
  TTestIndex = 1..100;
  TTestValue = -99..99;
  TTestList = array[TTestIndex] of TTestValue;
  PTestList = ^TTestList;
  TDate = class
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
    procedure SetDate(D, M, Y: Integer);
    function ShowDate: String;
  end;
  TMeasureData = record
    When: TDate;
    Count: TTestIndex;
    Data: PTestList;
  end;
  TMeasureList = array[1..50] of TMeasureData;
  TName = string[80];
  TSex = (Male, Female);
  PPersonData = ^TPersonData;
  TPersonData = record
    Name, FirstName: TName;
    Age: Integer;
    Married: Boolean;
    TFather, TChild, TSibling: PPersonData;
    case S: TSex of
      Male: (Bearded: Boolean);
      Female: (Pregnant: Boolean);
  end;
  TPersonBuf = array[0..SizeOf(TPersonData)-1] of Byte;
  TPeople = file of TPersonData;
```

In the example, *TRange*, *TNumber*, and *Integer* are identical types. *TTestIndex* is compatible and assignment-compatible with, but not identical to, the types *TNumber*, *TRange*, and *Integer*. Notice the use of constant expressions in the declarations of *TCharVal* and *TPersonBuf*.

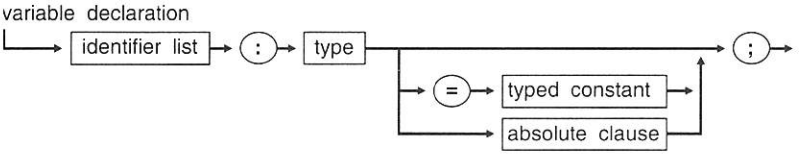
Chapter

5

Variables and typed constants

Variable declarations

A variable is an identifier that marks a value that can change. A *variable declaration* embodies a list of identifiers that designate new variables and their type.



The type given for the variable(s) can be a type identifier previously declared in a **type** declaration part in the same block, in an enclosing block, or in a unit; it can also be a new type definition.

Variables declared outside procedures and functions are called *global variables*. Variables declared within procedures and functions are called *local variables*.

When a variable declaration declares a single global variable, the declaration can optionally specify an *initial value* for the variable. If a global variable declaration does not explicitly specify an initial value, the memory occupied by the variable will initially be set to zero. The syntax of a *typed constant* used to specify the initial value of a global variable is explained in "Typed constants" on page 5-7.

It is not possible to specify an initial value for a local variable, and upon entry to a procedure or function, all local variables have undefined values.

When an identifier is specified within the identifier list of a variable declaration, that identifier is a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the block, unless the identifier is redeclared in an enclosed block. Redeclaration creates a new variable using the same identifier, without affecting the value of the original variable.

An example of a variable declaration part follows:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  C: Color;
  Done, Error: Boolean;
  Operator: (Plus, Minus, Times);
  Hue1, Hue2: set of Color;
  Today: Date;
  Results: MeasureList;
  P1, P2: Person;
  Matrix: array[1..10, 1..10] of Double;
```

Local variables and the stack

Variables declared within procedures and functions are called local variables, and reside in an application's *stack*. Each time a procedure or function is called, it allocates a set of local variables on the stack. On exit, the local variables are disposed of.

An application's stack is defined by two values: The *minimum stack size* and the *maximum stack size*. The values are controlled through the **$MINSTACKSIZE** and **$MAXSTACKSIZE** compiler directives, and default to 16,384 (16K) and 1,048,576 (1M) respectively. An application is guaranteed to always have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size.

If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If an application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated as needed in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an *EStackOverflow* exception is raised.

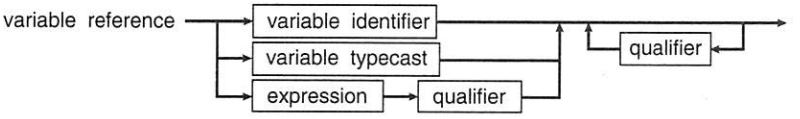
Note Delphi 1.0 and Borland Pascal supported a **$S** compiler directive to control stack-overflow checking. Stack-overflow checking is now completely automatic, but for backwards compatibility, the **$S** directive is still allowed.

Variable references

A variable reference signifies one of the following:

- A variable
- A component of a structured- or string-type variable
- A dynamic variable pointed to by a pointer-type variable

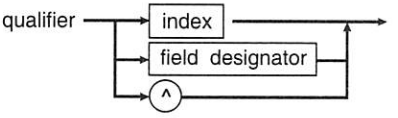
This is the syntax of a variable reference:



Note The syntax for a variable reference allows an expression that computes a pointer type value. The expression must be followed by a qualifier that dereferences the pointer value (or indexes the pointer value if the extended syntax is enabled with the {$X+} directive) to produce an actual variable reference.

Qualifiers

A variable reference can contain zero or more qualifiers that modify the meaning of the variable reference.



An array identifier with no qualifier, for example, references the entire array:

```
Results
```

An array identifier followed by an index denotes a specific component of the array—in this case, a structured variable:

```
Results[Current + 1]
```

With a component that is a record or object, the index can be followed by a field designator. Here the variable access signifies a specific field within a specific array component:

```
Results[Current + 1].Data
```

The field designator in a pointer field can be followed by the pointer symbol (^) to differentiate between the pointer field and the dynamic variable it points to:

```
Results[Current + 1].Data^
```

If the variable being pointed to is an array, indexes can be added to denote components of this array:

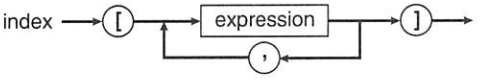
```
Results[Current + 1].Data^[J]
```

The pointer symbol is optional when dereferencing a structured type.

Arrays, strings, and indexes

A specific component of an array variable is denoted by a variable reference that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable is denoted by a variable reference that refers to the string variable, followed by an index that specifies the character position.



The index expressions select components in each corresponding dimension of the array. The number of expressions can't exceed the number of index types in the array declaration. Also, each expression's type must be assignment-compatible with the corresponding index type.

When indexing a multidimensional array, multiple indexes or multiple expressions within an index can be used interchangeably. For example,

```
Matrix[I][J]
```

is the same as

```
Matrix[I, J]
```

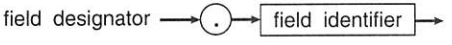
You can index a short string variable with a single index expression, whose value must be in the range 0..*N*, where *N* is the declared maximum length of the short string. The type of a character accessed through indexing of a short string is *Char*. The index of the first character in a string is 1. The element at index 0 contains the dynamic length of the string.

You can index a non-empty long string variable with a single index expression, whose value must be in the range 1..*N*, where *N* is the dynamic length of the long string. The type of a character accessed through indexing of a long string is *Char*. The index of the first character in a long string is 1.

A value of type *PChar*, *PAnsiChar*, or *PWideChar* can be indexed with a single index expression of type *Integer*. The index expression specifies an offset (number of characters or wide characters) to add to the character pointer before it is dereferenced to produce a *Char*, *AnsiChar*, or *WideChar* type variable reference.

Records and field designators

A specific field of a record variable is denoted by a variable reference that refers to the record variable, followed by a field designator specifying the field.



These are examples of a field designator:

```
Today.Year
Results[1].Count
Results[1].When.Month
```

In a statement within a **with** statement, a field designator doesn't have to be preceded by a variable reference to its containing record.

Object component designators

The format of an object component designator is the same as that of a record field designator; that is, it consists of an instance (a variable reference), followed by a period and a component identifier. A component designator that designates a method is called a *method designator*. A **with** statement can be applied to an instance of a class type. In that case, the instance and the period can be omitted in referencing components of the class type.

The instance and the period can also be omitted within any method block, and when they are, the effect is the same as if *Self* and a period were written before the component reference.

Pointers and dynamic variables

The value of a pointer variable is either **nil** or the address of a dynamic variable.

The dynamic variable pointed to by a pointer variable is referenced by writing the pointer symbol (^) after the pointer variable.

You create dynamic variables and their pointer values with the procedures *New* and *GetMem*. You can use the @ (address-of) operator and the function *Ptr* to create pointer values that are treated as pointers to dynamic variables.

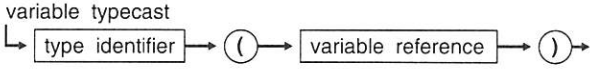
nil doesn't point to any variable. The results are undefined if you access a dynamic variable when the pointer's value is **nil** or undefined.

These are examples of references to dynamic variables:

```
P1^
P1^.Sibling^
Results[1].Data^
```

Variable typecasts

Variable typecasting changes the variable reference of one type into a variable reference of another type. The programmer is responsible for determining the validity of a typecast.



When a variable typecast is applied to a variable reference, the variable reference is treated as an instance of the type specified by the type identifier. The size of the variable must be the same as the size of the type denoted by the type identifier.

A variable typecast can be followed by one or more qualifiers, as allowed by the specific type.

Some examples of variable typecasts follow:

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
```

```
  TWordRec = record
    Low, High: Word;
  end;
  PByte = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;
begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end.
```

Notice the use of the *TByteRec* type to access the low- and high-order bytes of a word. This corresponds to the built-in functions *Lo* and *Hi*, except that a variable typecast can also be used on the left side of an assignment. Also, observe the use of the *TWordRec* type to access the low- and high-order words of a long integer.

Object Pascal fully supports variable typecasts involving procedural types. For example, given the declarations

```
type
  Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

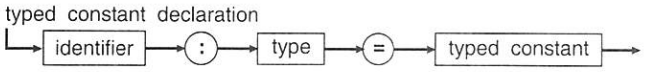
you can construct the following assignments:

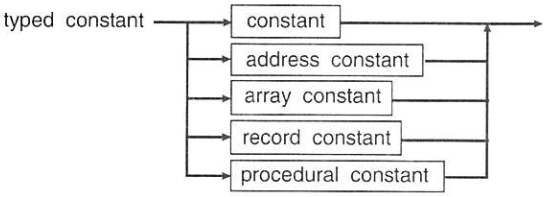
```
F := Func(P);           { Assign procedural value in P to F }
Func(P) := F;           { Assign procedural value in F to P }
@F := P;                { Assign pointer value in P to F }
P := @F;                { Assign pointer value in F to P }
N := F(N);              { Call function via F }
N := Func(P)(N);        { Call function via P }
```

In particular, notice that the address operator (@), when applied to a procedural variable, can be used on the left side of an assignment. Also, notice the typecast on the last line to call a function via a pointer variable.

Typed constants

The declaration of a typed constant corresponds to the declaration of a *read-only* variable. Typed constants can be used exactly like variables of the same type, except that they cannot be modified.





In addition to a normal constant expression, the value of a typed constant can be specified using a *constant-address expression*. A constant-address expression is an expression that involves taking the address, offset, or segment of a global variable, a typed constant, a procedure, or a function. Constant-address expressions can't reference local variables (stack-based) or dynamic (heap-based) variables, because their addresses can't be computed at compile time.

Note The **$J** compiler directive allows the declaration of typed constants that can be modified. Typed constants declared in the {**$J–**} state are read-only and cannot be modified. For backwards compatibility with previous versions of Delphi and Borland Pascal, typed constants declared in the {**$J+**} state can be modified, and are in essence initialized variables. For new applications, use of the {**$J+**} state is not recommended.

Simple-type constants

Declaring a typed constant as a simple type specifies the value of the constant:

```
const
  Maximum: Integer = 9999;
  Factor: Double = -0.1;
  Breakchar: Char = #3;
```

Because a typed constant is actually a variable with a constant value, it can't be interchanged with ordinary constants. For example, it can't be used in the declaration of other constants or types:

```
const
  Min: Integer = 0;
  Max: Integer = 99;
type
  Vector = array[Min..Max] of Integer;
```

The *Vector* declaration is invalid because *Min* and *Max* are typed constants.

String-type constants

Declaration of a typed constant of a string type simply specifies the string constant:

```
const
  Heading: string = 'Section';
  NewLine: string = #13#10;
  TrueStr: string = 'Yes';
  FalseStr: string = 'No';
```

The typed constants in the above declarations are all of type long string. To declare a short string typed constant, include a length specifier in the declaration:

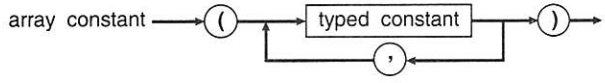
```
const
  ShortStr: string[5] = 'Short';
```

Structured-type constants

The declaration of a structured-type constant specifies the value of each of the structure's components. Object Pascal supports the declaration of array, record, and set type constants. File type constants and constants of array, and record types that contain file type components aren't allowed.

Array-type constants

The declaration of an array-type constant, enclosed in parentheses and separated by commas, specifies the values of the components.



This is an example of an array-type constant:

```
type
  TStatus = (Active, Passive, Waiting);
  TStatusMap = array[TStatus] of string;
const
  StatStr: TStatusMap = ('Active', 'Passive', 'Waiting');
```

This example defines the array constant *StatStr*, which can be used to convert values of type *TStatus* into their corresponding string representations. These are the components of *StatStr*:

```
StatStr[Active] = 'Active'
StatStr[Passive] = 'Passive'
StatStr[Waiting] = 'Waiting'
```

The component type of an array constant can be any type except a file type. Packed string-type constants (character arrays) can be specified both as single characters and as strings. The definition

```
const
  Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5',
    '6', '7', '8', '9');
```

can be expressed more conveniently as

```
const
  Digits: array[0..9] of Char = '0123456789';
```

A zero-based character array can be initialized with a string that is shorter than the declared length of the array. For example,

```
var
  FileName: array[0..79] of Char = 'TEST.PAS';
```

In such cases, the remaining characters are set to null (#0) and the array effectively contains a null-terminated string. For more about null-terminated strings, see Chapter 17, "Using null-terminated strings."

Multidimensional-array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas. The innermost constants correspond to the rightmost dimensions. The declaration

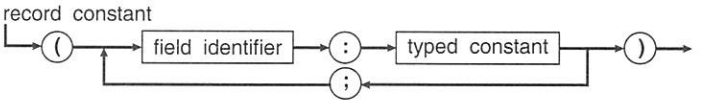
```
type
  TCube = array[0..1, 0..1, 0..1] of Integer;
const
  Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

provides an initialized array *Maze* with the following values:

```
Maze[0, 0, 0] = 0
Maze[0, 0, 1] = 1
Maze[0, 1, 0] = 2
Maze[0, 1, 1] = 3
Maze[1, 0, 0] = 4
Maze[1, 0, 1] = 5
Maze[1, 1, 0] = 6
Maze[1, 1, 1] = 7
```

Record-type constants

The declaration of a record-type constant specifies the identifier and value of each field, enclosed in parentheses and separated by semicolons.



Some examples of record constants follow:

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of Point;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
    Nov, Dec);
  TDate = record
    D: 1..31;
    M: Month;
    Y: 1900..1999;
```

```
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

The fields must be specified in the same order as they appear in the definition of the record type. If a record contains fields of file types, constants of that record type can't be declared. If a record contains a variant, only fields of the selected variant can be specified. If the variant contains a tag field, then its value must be specified.

Set-type constants

Just like a simple-type constant, the declaration of a set-type constant specifies the value of the set using a constant expression. Here are some examples:

```
type
  TDigits = set of 0..9;
  TLetters = set of 'A'..'Z';
const
  EvenDigits: TDigits = [0, 2, 4, 6, 8];
  Vowels: TLetters = ['A', 'E', 'I', 'O', 'U', 'Y'];
  HexDigits: set of '0'..'z' = ['0'..'9', 'A'..'F', 'a'...f'];
```

Pointer-type constants

The declaration of a pointer-type constant uses a constant-address expression to specify the pointer value. Some examples follow:

```
type
  TDirection = (Left, Right, Up, Down);
  PNode = ^TNode;
  TNode = record
    Next: PNode;
    Symbol: string;
    Value: TDirection;
  end;
const
  N1: TNode = (Next: nil; Symbol: 'DOWN'; Value: Down);
  N2: TNode = (Next: @N1; Symbol: 'UP'; Value: Up);
  N3: TNode = (Next: @N2; Symbol: 'RIGHT'; Value: Right);
  N4: TNode = (Next: @N3; Symbol: 'LEFT'; Value: Left);
var
  DirectionList: PNode = @N4;
```

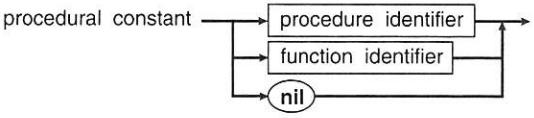
A typed constant of type *PChar* can be initialized with a string constant. For example,

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

The result is that the pointer now points to an area of memory that contains a zero-terminated copy of the string literal. See Chapter 17, "Using null-terminated strings," for more information.

Procedural-type constants

A procedural-type constant must specify the identifier of a procedure or function that is assignment-compatible with the type of the constant, or it must specify the value **nil**.



Here's an example:

```
type
  TErrorProc = procedure(ErrorCode: Integer);

procedure DefaultError(ErrorCode: Integer);
begin
  Writeln('Error ', ErrorCode, '.');
end;

const
  ErrorHandler: TErrorProc = DefaultError;
```

Chapter

6

Expressions

Expressions are made up of *operators* and *operands*. Most Object Pascal operators are *binary*; they take two operands. The rest are *unary* and take only one operand. Binary operators use the usual algebraic form (for example, $A + B$). A unary operator always precedes its operand (for example, $-B$).

In more complex expressions, rules of precedence clarify the order in which operations are performed.

Table 6.1 Precedence of operators

Operators	Precedence	Categories
@, not	first (high)	unary operators
***, /, div, mod, and, shl, shr, as**	second	multiplying operators
+,–, or, xor	third	adding operators
=, <>, <, >, <=, >=, in, is	fourth (low)	relational operators

There are three basic rules of precedence:

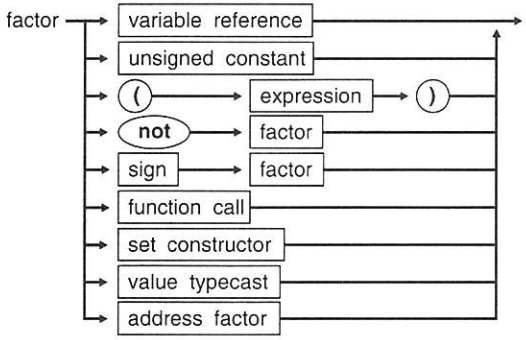
- An operand between two operators of different precedence is bound to the operator with higher precedence.
- An operand between two equal operators is bound to the one on its left.
- Expressions within parentheses are evaluated prior to being treated as a single operand

Operations with equal precedence are normally performed from left to right, although the compiler may rearrange the operands to generate optimum code.

Expression syntax

The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple expressions.

A factor's syntax follows:



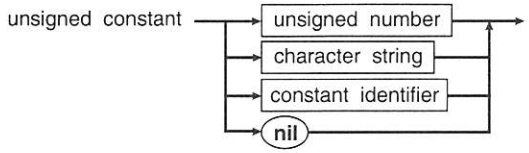
A function call activates a function and denotes the value returned by the function. See "Function calls" on page 6-10.

A set constructor denotes a value of a set type. See "Set constructors" on page 6-11.

A value typecast changes the type of a value. See "Value typecasts" on page 6-12.

An address factor computes the address of a variable, procedure, function, or method. See "The @ operator" on page 6-10.

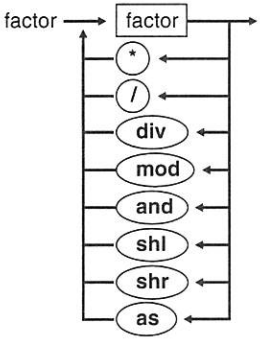
An unsigned constant has the following syntax:



These are some examples of factors:

```
X                          { Variable reference }
@X                         { Pointer to a variable }
15                         { Unsigned constant }
(X + Y + Z)                { Subexpression }
Sin(X / 2)                 { Function call }
exit['0'..'9', 'A'..'Z']   { Set constructor }
not Done                   { Negation of a Boolean }
Char(Digit + 48)           { Value typecast }
```

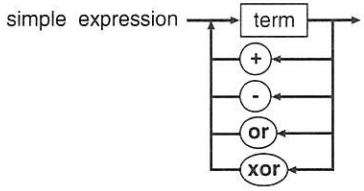
Terms apply the multiplying operators to factors:



Here are some examples of terms:

```
X * Y
Z / (1 - Z)
Y shl 2
(X <= Y) and (Y < Z)
```

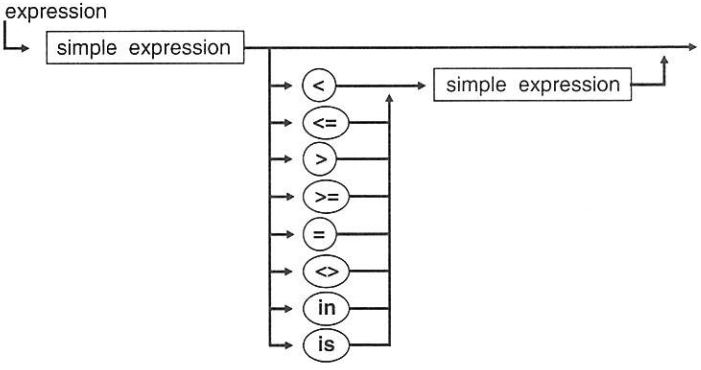
Simple expressions apply adding operators and signs to terms:



Here are some examples of simple expressions:

```
X + Y
-X
Hue1 + Hue2
I * J + 1
```

An expression applies the relational operators to simple expressions:



Here are some examples of expressions:

```
X = 1.5
Done <> Error
(I < J) = (J < K)
C in Hue1
```

Operators

Operators are classified as arithmetic operators, logical operators, string operators, character-pointer operators, set operators, relational operators, and the @ operator.

Arithmetic operators

Table 6.2 and Table 6.3 show the types of operands and results for binary and unary arithmetic operations.

Table 6.2 Binary arithmetic operations

Operator	Operation	Operand types	Result type
+	addition	integer type	integer type
		real type	real type
–	subtraction	integer type	integer type
		real type	real type
*	multiplication	integer type	integer type
		real type	real type
/	division	integer type	real type
		real type	real type
div	integer division	integer type	integer type
mod	remainder	integer type	integer type

The + operator is also used as a string or set operator, and the +, –, and * operators are also used as set operators.

Table 6.3 Unary arithmetic operations

Operator	Operation	Operand types	Result type
+	sign identity	integer type	integer type
		real type	real type
–	sign negation	integer type	integer type
		real type	real type

Any operand whose type is a subrange of an ordinal type is treated as if it were of the ordinal type.

If both operands of a +, –, *, **div, or mod** operator are of an integer type, the result type is of the common type of the two operands. For a definition of common types, see "Ordinal types" on page 4-2.

If the operand of the sign identity or sign negation operator is of an integer type, the result is of the same integer type. If the operator is of a real type, the type of the result is *Extended*.

The value of *X* / *Y* is always of type *Extended* regardless of the operand types. A runtime error occurs if *Y* is zero.

The value of *I* **div** *J* is the mathematical quotient of *I* / *J*, rounded in the direction of zero to an integer-type value. A runtime error occurs if *J* is zero.

The **mod** operator returns the remainder obtained by dividing its two operands; that is,

```
I mod J = I - (I div J) * J
```

The sign of the result of **mod** is the same as the sign of *I*. A runtime error occurs if *J* is zero.

Logical operators

The types of operands and results for logical operations are shown in Table 6.4.

Table 6.4 Logical operations

Operator	Operation	Operand types	Result type
not	bitwise negation	integer type	*Boolean*
and	bitwise and	integer type	*Boolean*
or	bitwise or	integer type	*Boolean*
xor	bitwise xor	integer type	*Boolean*
shl	bitwise shift left	integer type	*Boolean*
shr	bitwise shift right	integer type	*Boolean*

If the operand of the **not** operator is of an integer type, the result is of the same integer type. The **not** operator is a unary operator.

If both operands of an **and, or, or xor** operator are of an integer type, the result type is the common type of the two operands.

The operations *I* **shl** *J* and *I* **shr** *J* shift the value of *I* to the left right by *J* bits. The result type is the same as the type of *I*.

Boolean operators

The types of operands and results for Boolean operations are shown in Table 6.5.

Table 6.5 Boolean operations

Operator	Operation	Operand types	Result type
not	negation	Boolean type	*Boolean*
and	logical and	Boolean type	*Boolean*
or	logical or	Boolean type	*Boolean*
xor	logical xor	Boolean type	*Boolean*

Normal Boolean logic governs the results of these operations. For instance, *A* **and** *B* **is** *True* only if both *A* and *B* are *True*.

Object Pascal supports two different models of code generation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation.

Complete evaluation means that every operand of a Boolean expression built from the **and** and **or** operators is guaranteed to be evaluated, even when the result of the entire expression is already known. This model is convenient when one or more operands of an expression are functions with side effects that alter the meaning of the program.

Short-circuit evaluation guarantees strict left-to-right evaluation and that evaluation stops as soon as the result of the entire expression becomes evident. This model is convenient in most cases because it guarantees minimum execution time, and usually minimum code size. Short-circuit evaluation also makes possible the evaluation of constructs that would not otherwise be legal. For example,

```
while (I <= Length(S)) and (S[I] <> ' ') do
  Inc(I);
while (P <> nil) and (P^.Value <> 5) do
  P := P^.Next;
```

In both cases, the second test isn't evaluated if the first test is *False*.

The evaluation model is controlled through the **$B** compiler directive. The default state is {**$B–**}, and in this state, the compiler generates short-circuit evaluation code. In the {**$B+**} state, the compiler generates complete evaluation.

String operator

The types of operands and results for string operation are shown in Table 6.6.

Table 6.6 String operation

Operator	Operation	Operand types	Result type
+	concatenation	string type, *Char* type, or packed string type	string type

Object Pascal allows the + operator to be used to concatenate two string operands. The result of the operation *S* + *T*, where *S* and *T* are of a string type, a *Char* type, or a

packed string type, is the concatenation of *S* and *T*. The result is compatible with any string type (but not with *Char* types and packed string types). If both *S* and *T* are short strings and the resulting string is longer than 255 characters, it's truncated after character 255. If either *S* or *T* is a long string, the result is also a long string.

Character-pointer operators

The plus (+) and minus (–) operators can be used to increment and decrement the offset part of a character pointer value, and the minus operator can be used to calculate the distance (difference) between the offset parts of two character pointers. Assuming that *P* and *Q* are values of type *PChar* and *I* is a value of type *Integer*, these constructs are allowed:

Table 6.7 Permitted PChar constructs

Operation	Result
P + *I*	Add *I* to *P*
I + *P*	Add *I* to *P*
P – *I*	Subtract *I* from *P*
P – *Q*	Subtract *Q* from *P*

The operations *P* + *I* and *I* + *P* adds *I* to the address given by *P*, producing a pointer that points *I* characters after *P*. The operation *P* – *I* subtracts *I* from the address given by *P*, producing a pointer that points *I* characters before *P*.

The operation *P* – *Q* computes the distance between *Q (the lower address)* and *P (the higher address)*, resulting in a value of type *Integer* that gives the number of characters between *Q* and *P*. This operation assumes that *P* and *Q* point within the same character array. If the two character pointers point into different character arrays, the result is undefined.

Set operators

The types of operands for set operations are shown in Table 6.6.

Table 6.8 Set operations

Operator	Operation	Operand types
+	union	compatible set types
–	difference	compatible set types
*	intersection	compatible set types

The results of set operations conform to the rules of set logic:

- An ordinal value *C* is in *A* + *B* only if *C* is in *A* or *B*.
- An ordinal value *C* is in *A* – *B* only if *C* is in *A* and not in *B*
- An ordinal value *C* is in *A* * *B* only if *C* is in both *A* and *B*.

If the smallest ordinal value that is a member of the result of a set operation is *A* and the largest is *B*, then the type of the result is **set of** *A..B*.

Variant operators

The +, –, *, /, **div**, **mod**, **shl**, **shr**, **and**, **or**, **xor**, and **not** operators support operands of type *Variant*. For binary operators, if one operand is of type *Variant*, the other operand is automatically converted to type *Variant*. For further information on the *Variant* type and operations on variants, see "Variant types" on page 4-20.

Relational operators

The types of operands and results for relational operations are shown in Table 6.9.

Table 6.9 Relational operations

Operator type	Operation	Operand types	Result type
=	equal	compatible simple, class, class reference, pointer, set, string, packed string, or variant types	*Boolean*
<>	not equal	compatible simple, class, class reference, pointer, set, string, packed string, or variant types	*Boolean*
<	less than	compatible simple, string, packed string, *PChar*, or variant types	*Boolean*
>	greater than	compatible simple, string, packed string, *PChar*, or variant types	*Boolean*
<=	less than or equal to	compatible simple, string, packed string, *PChar*, or variant types	*Boolean*
>=	greater than or equal to	compatible simple, string, packed string, *PChar*, or variant types	*Boolean*
<=	subset of	compatible set types	*Boolean*
>=	superset of	compatible set types	*Boolean*
in	member of	left operand, any ordinal type *T*; right operand, set whose base is compatible with *T*	*Boolean*

Comparing simple types

When the operands =, <>, <, >, >=, or <= are of simple types, they must be compatible types; however, if one operand is of a real type, the other can be of an integer type.

Comparing strings

The relational operators =, <>, <, >>, >=, and <= compare strings according to the ordering of the extended ASCII character set. Any two string values can be compared because all string values are compatible.

A character-type value is compatible with a string-type value. When the two are compared, the character-type value is treated as a string-type value with length 1.

When a packed string-type value with *N* components is compared with a string-type value, it's treated as a string-type value with length *N*.

Comparing packed strings

The relational operators =, <>, <, >, >=, and <= can also be used to compare two packed string-type values if both have the same number of components. If the number of components is N, then the operation corresponds to comparing two strings, each of length N.

Comparing pointers and references

The operators = and <> can be used on compatible pointer-type, class-type, class-reference-type operands. Two pointers are equal only if they point to the same object.

Comparing character pointers

The >, <, >=, and <= operators can be applied to *PChar* values. Note, however, that these relational tests assume that the two pointers being compared point within the same character array. If the two character pointers point into different character arrays, the result is undefined.

Comparing variants

The rules used for comparing values of type *Variant* are described in "Variant types" on page 4-20.

Comparing sets

If *A* and *B* are set operands, their comparisons produce these results:

- *A* = *B* is *True* only if *A* and *B* contain exactly the same members; otherwise, *A* <> *B*.
- *A* <= *B* is *True* only if every member of *A* is also a member of *B*.
- *A* >= *B* is *True* only if every member of *B* is also a member of *A*.

Testing set membership

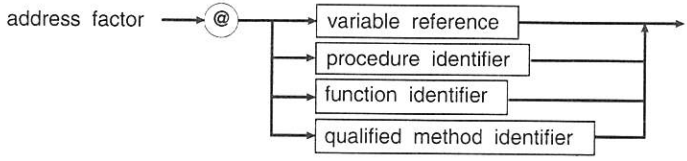
The **in** operator returns *True* when the value of the ordinal-type operand is a member of the set-type operand; otherwise, it returns *False*.

Class operators

Object Pascal defines two operators, **is** and **as**, that operate on class and object references. See Chapter 10, "Class types".

The @ operator

The @ operator is used in an address factor to compute the address of a variable, procedure, function, or method.



The @ operator returns the address of its operand, that is, it constructs a pointer value that points to the operand.

@ with a variable

When applied to a variable reference, @ returns a pointer to the variable. The type of the resulting pointer value is controlled through the **$T** compiler directive: In the {**$T–**} state (the default), the result type is *Pointer*. In other words, the result is an untyped pointer, which is compatible with all other pointer types. In the {**$T+**} state, the type of the result is ^*T*, where *T* is the type of the variable reference. In other words, the result is of a type that is compatible only with other pointers to the type of the variable.

Special rules apply to use of the @ operator with a procedural variable. For more details, see "Procedural types in expressions" on page 6-12.

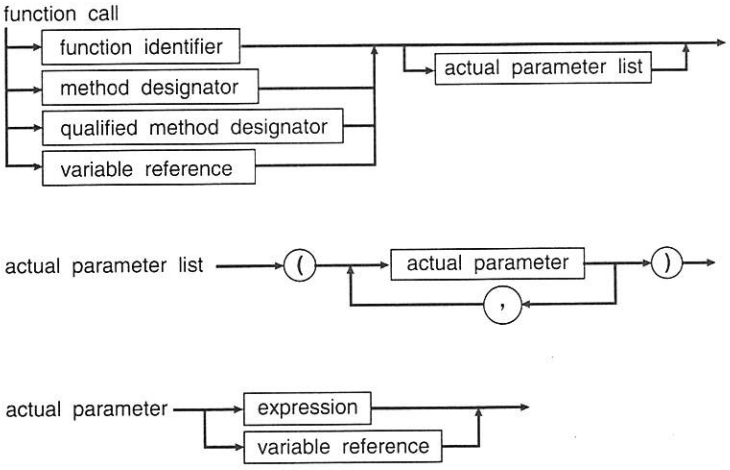
@ with a procedure, function, or method

You can apply @ to a procedure, function, or method to produce a pointer to the routine's entry point. The type of the resulting pointer is always *Pointer*, regardless of the state of the **$T** compiler directive. In other words, the result is always an untyped pointer, which is compatible with all other pointer types.

When @ is applied to a method, the method must be specified through a *qualified-method identifier* (a class-type identifier, followed by a period, followed by a method identifier).

Function calls

A function call activates a function specified by a function identifier, a method designator, a qualified-method designator, or a procedural-type variable reference. The function call must have a list of actual parameters if the corresponding function declaration contains a list of formal parameters. Each parameter takes the place of the corresponding formal parameter according to parameter rules explained in Chapter 9, "Procedures and functions."



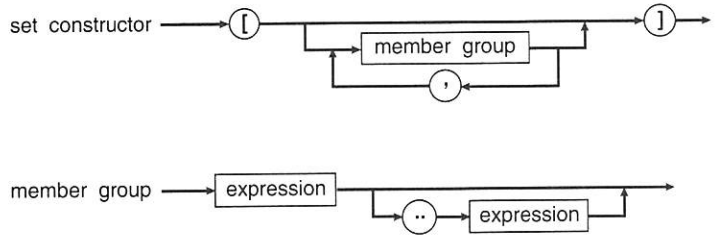
These are some examples of function calls:

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
MakeBeep();
```

Object Pascal allows the result of a function call to be discarded, in essence treating the function call as a procedure statement. See "Procedure statements" on page 7-2.

Set constructors

A set constructor denotes a set-type value, and is formed by writing expressions within brackets ([]). Each expression denotes a value of the set.



The notation [ ] denotes the empty set, which is assignment-compatible with every set type. Any member group *X*..*Y* denotes as set members all values in the range *X*..*Y*. If *X* is greater than *Y*, then *X*..*Y* doesn't denote any members and [*X*..*Y*] denotes the empty set.

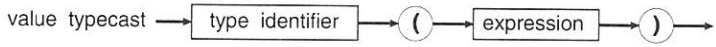
All expression values in member groups in a particular set constructor must be of the same ordinal type.

These are some examples of set constructors:

```
[red, C, green]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

Value typecasts

The type of an expression can be changed to another type through a value typecast.



The expression type and the specified type must both be either ordinal types or pointer types. For ordinal types, the resulting value is obtained by converting the expression. The conversion may involve truncation or extension of the original value if the size of the specified type is different from that of the expression. In cases where the value is extended, the sign of the value is always preserved; that is, the value is sign-extended.

The syntax of a value typecast is almost identical to that of a variable typecast. Value typecasts operate on values, however, not on variables, and therefore they can't participate in variable references; that is, a value typecast can't be followed by qualifiers. In particular, value typecasts can't appear on the left side of an assignment statement. See "Variable typecasts" on page 5-5.

These are some examples of value typecasts:

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

Procedural types in expressions

Usually, using a procedural variable in a statement or an expression calls the procedure or function stored in the variable. There is one exception: When the compiler sees a procedural variable on the left side of an assignment statement, it knows that the right side has to represent a procedural value. For example, consider the following program:

```
type
  IntFunc = function: Integer;

var
  F: IntFunc;
  N: Integer;

function ReadInt: Integer; far;
```

```
var
  I: Integer;
begin
  Read(I);
  ReadInt := I;
end;

begin
  F := ReadInt; { Assign procedural value }
  N := ReadInt; { Assign function result }
end.
```

The first statement in the main program assigns the procedural value (address of) ReadInt to the procedural variable *F*, where the second statement calls *ReadInt* and assigns the returned value to *N*. The distinction between getting the procedural value or calling the function is made by the type of the variable being assigned (*F* or *N*).

Unfortunately, there are situations where the compiler can't determine the desired action from the context. For example, in the following statement there is no obvious way the compiler can know if it should compare the procedural value in *F* to the procedural value of *ReadInt* to determine if *F* currently points to *ReadInt*, or if it should call *F* and *ReadInt* and then compare the returned values.

```
if F = ReadInt then
  Edit1.Text := 'Equal';
```

Object Pascal syntax, however, specifies that the occurrence of a function identifier in an expression denotes a call to that function, so the effect of the preceding statement is to call *F* and *ReadInt*, and then compare the returned values. To compare the procedural value in *F* to the procedural value of *ReadInt*, the following construct must be used:

```
if @F = @ReadInt then
  Edit1.Text := 'Equal';
```

When applied to a procedural variable or a procedure or function identifier, the address (@) operator prevents the compiler from calling the procedure, and at the same time converts the argument into a pointer. *@F* converts *F* into an untyped pointer variable that contains an address, and *@ReadInt* returns the address of *ReadInt*; the two pointer values can then be compared to determine if *F* currently refers to *ReadInt*.

The @ operator is often used when assigning an untyped pointer value to a procedural variable. For example, the *GetProcAddress* function defined by Windows (in the *WinProcs* unit) returns the address of an exported function in a DLL as an untyped pointer value. Using the @ operator, the result of a call to *GetProcAddress* can be assigned to a procedural variable:

```
type
  TStrComp = function(Str1, Str2: Pchar): Integer;
var
  StrComp: TStrComp;
  .
  .
  .
begin
  .
  .
  .
  @StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

```
  :
end.
```

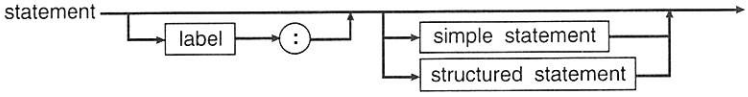
Note To get the memory address of a procedural variable rather than the address stored in it, use a double address (@@) operator. For example, where *@P* means convert *P* into an untyped pointer variable, *@@P* means return the physical address of the variable *P*.

Chapter

7

Statements

Statements describe algorithmic actions that can be executed. Labels can prefix statements, and these labels can be referenced by **goto** statements.

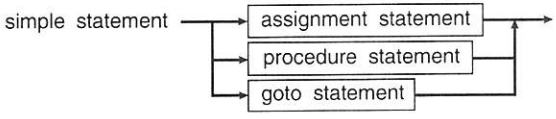


A label is either a digit sequence in the range 0 to 9999 or an identifier.

There are three main types of statements: simple statements and structured statements.

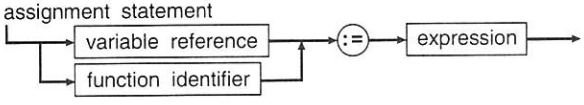
Simple statements

A simple statement is a statement that doesn't contain any other statements.



Assignment statements

Assignment statements replace the current value of a variable with a new value specified by an expression. They can be used to set the return value of the function also.



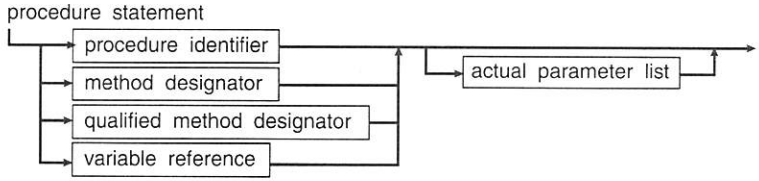
The expression must be assignment-compatible with the type of the variable or the type of the function result. See "Type compatibility" on page 4-32.

Some examples of assignment statements follow:

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Huel := [Blue, Succ(C)];
I := Sqr(J) - I * K;
```

Procedure statements

A procedure statement activates a procedure specified by a procedure identifier, a method designator, a qualified-method designator, or a procedural-type variable reference. If the corresponding procedure declaration contains a list of formal parameters, then the procedure statement must have a matching list of actual parameters (parameters listed in definitions are *formal parameters*; in the calling statement, they are *actual parameters)*. The actual parameters are passed to the formal parameters as part of the call. For more information, see Chapter 9, "Procedures and functions."

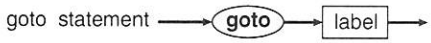


Some examples of procedure statements follow:

```
PrintHeading;
Transpose(A, N, M);
Find(Name, Address);
DoStuff();
```

Goto statements

A **goto** statement transfers program execution to the statement marked by the specified label. The syntax diagram of a **goto** statement follows:



When using goto statements, observe the following rules:

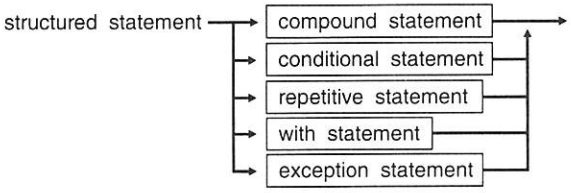
- The label referenced by a **goto** statement must be in the same block as the **goto** statement. In other words, it's not possible to jump into or out of a procedure or function.
- Jumping into a structured statement from outside that structured statement (that is, jumping to a deeper level of nesting) can have undefined effects, although the

compiler doesn't indicate an error. For example, you shouldn't jump into the middle of a **for** loop.

Good programming practices recommend that you use **goto** statements as little as possible.

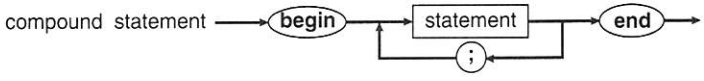
Structured statements

Structured statements are constructs composed of other statements that are to be executed in sequentially (compound and **with** statements), conditionally (conditional statements), or repeatedly (repetitive statements).



Compound statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The component statements are treated as one statement, crucial in contexts where the Object Pascal syntax only allows one statement. **begin** and **end** bracket the statements, which are separated by semicolons.

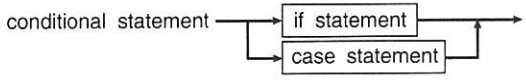


Here's an example of a compound statement:

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

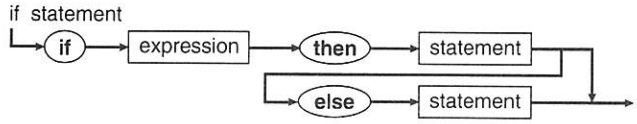
Conditional statements

A conditional statement selects for execution a single one (or none) of its component statements.



If statements

The syntax for an **if** statement reads like this:



The expression must yield a result of the standard type *Boolean*. If the expression produces the value *True*, then the statement following **then** is executed.

If the expression produces *False* and the **else** part is present, the statement following **else** is executed; if the **else** part isn't present, execution continues at the next statement following the **if** statement.

The syntactic ambiguity arising from the construct

```
if e1 then if e2 then s1 else s2;
```

is resolved by interpreting the construct as follows:

Note No semicolon is allowed preceding an **else** clause.

```
if e1 then
begin
  if e2 then
    s1
  else
    s2
end;
```

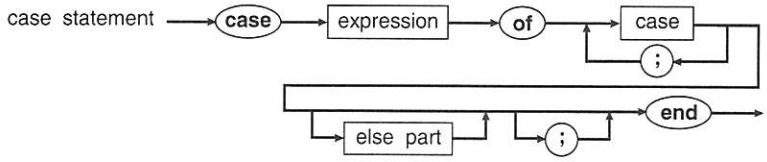
Usually, an **else** is associated with the closest **if** not already associated with an **else**.

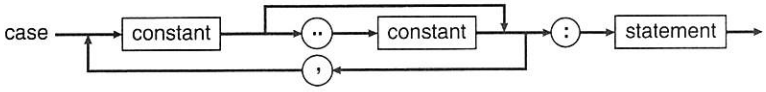
Two examples of **if** statements follow:

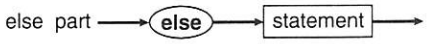
```
if X < 1.5 then
  Z := X + Y
else
  Z := 1.5;
if P1 <> nil then
  P1 := P1^.Father;
```

Case statements

The **case** statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called *case constants*) or with the word **else**. The selector must be of an ordinal type, so string types are invalid selector types. All **case** constants must be unique and of an ordinal type compatible with the selector type.







The **case** statement executes the statement prefixed by a **case** constant equal to the value of the selector or a **case** range containing the value of the selector. If no such **case** constant of the **case** range exists and an **else** part is present, the statement following **else** is executed. If there is no **else** part, execution continues with the next statement following the **if** statement.

These are examples of **case** statements:

```
case Operator of
  Plus: X := X + Y;
  Minus: X := X - Y;
  Times: X := X * Y;
end;
```

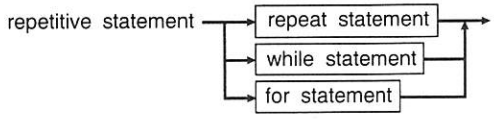
```
case I of
  0, 2, 4, 6, 8: Edit1.Text := 'Even digit';
  1, 3, 5, 7, 9: Edit1.Text := 'Odd digit';
  10..100: Edit1.Text := 'Between 10 and 100';
else
  Edit1.Text := 'Negative or greater than 100';
end;
```

Ranges in case statements must not overlap. So for example, the following case statement is not allowed:

```
case MySelector of
  5: Edit1.Text := 'Special case';
  1..10: Edit1.Text := 'General case';
end;
```

Repetitive statements

Repetitive statements specify certain statements to be executed repeatedly.



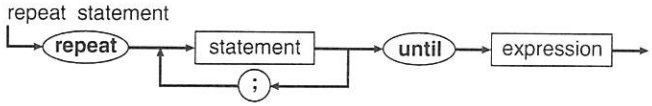
If the number of repetitions is known beforehand, the **for** statement is the appropriate construct. Otherwise, the **while** or **repeat** statement should be used.

The *Break* and *Continue* standard procedures can be used to control the flow of repetitive statements: *Break* terminates a repetitive statement, and *Continue* continues

with the next iteration of a repetitive statement. For more details on these standard procedures, see the *Visual Component Library Reference*.

Repeat statements

A **repeat** statement contains an expression that controls the repeated execution of a statement sequence within that **repeat** statement.



The expression must produce a result of type *Boolean*. The statements between the symbols **repeat** and **until** are executed in sequence until, at the end of a sequence, the expression yields *True*. The sequence is executed at least once because the expression is evaluated *after* the execution of each sequence.

These are examples of **repeat** statements:

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While statements

A **while** statement contains an expression that controls the repeated execution of a statement (which can be a compound statement).

The expression controlling the repetition must be of type *Boolean*. It is evaluated *before* the contained statement is executed. The contained statement is executed repeatedly as long as the expression is *True*. If the expression is *False* at the beginning, the statement isn't executed at all.

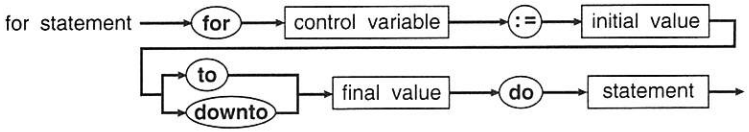
These are examples of **while** statements:

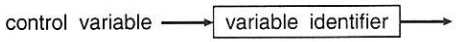
```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InFile) do
begin
  Readln(InFile, Line);
```

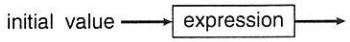
```
    Process(Line);
  end;
```

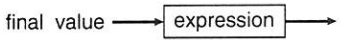
For statements

The **for** statement causes a statement to be repeatedly executed while a progression of values is assigned to a control variable. Such a statement can be a compound statement.









The control variable must be a variable identifier (without any qualifier) that is local in scope to the block containing the **for** statement. The control variable must be of an ordinal type. The initial and final values must be of a type assignment-compatible with the ordinal type. See Chapter 8, "Blocks, locality, and scope" for a discussion of locality and scope.

When a **for** statement is entered, the initial and final values are determined once for the remainder of the execution of the **for** statement.

The statement contained by the **for** statement is executed once for every value in the range *initial value* to *final value*. The control variable always starts off at *initial value*. When a **for** statement uses **to**, the value of the control variable is incremented by one for each repetition. If *initial value* is greater than *final value*, the contained statement isn't executed. When a **for** statement uses **downto**, the value of the control variable is decremented by one for each repetition. If *initial value* value is less than *final value*, the contained statement isn't executed.

If the contained statement alters the value of the control variable, your results will probably not be what you expect. After a **for** statement is executed, the value of the control variable value is undefined, unless execution of the **for** statement was interrupted by a **goto** from the **for** statement.

With these restrictions in mind, the **for** statement

```
for V := Expr1 to Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
```

```
    Temp2 := Expr2;
    if Temp1 <= Temp2 then
    begin
      V := Temp1;
      Body;
      while V <> Temp2 do
      begin
        V := Succ(V);
        Body;
      end;
    end;
end;
```

and the **for** statement

```
for V := Expr1 downto Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 >= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Pred(V);
      Body;
    end;
  end;
end;
```

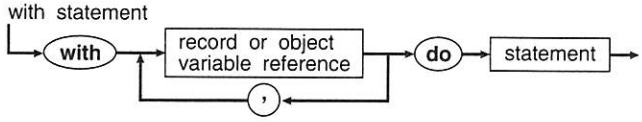
where *Temp1* and *Temp2* are auxiliary variables of the host type of the variable *V* and don't occur elsewhere in the program.

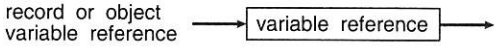
These are examples of **for** statements:

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I]

for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Check(C);
```

With statements

The **with** statement is shorthand for referencing the fields of a record, and the fields and methods of an object. Within a **with** statement, the fields of one or more specific record variables can be referenced using their field identifiers only. The syntax of a **with** statement follows:





Given this type declaration,

```
type
   TDate = record
      Day: Integer;
      Month: Integer;
      Year: Integer;
   end;

var
  OrderDate: TDate;
```

here is an example of a **with** statement:

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1
  end
  else
    Month := Month + 1;
```

This is equivalent to

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

Within a **with** statement, each variable reference is first checked to see if it can be interpreted as a field of the record. If so, it's always interpreted as such, even if a variable with the same name is also accessible. Suppose the following declarations have been made:

```
type
  TPoint = record
```

```
    X, Y: Integer;
  end;
var
  X: TPoint;
  Y: Integer;
```

In this case, both *X* and *Y* can refer to a variable or to a field of the record. In the statement

```
with X do
begin
  X := 10;
  Y := 25;
end;
```

the *X* between **with** and **do** refers to the variable of type *TPoint*, but in the compound statement, *X* and *Y* refer to *X.X* and *X.Y*.

The statement

```
with V1, V2, ... Vn do S;
```

is equivalent to

```
with V1 do
  with V2 do
   :
     with Vn do
       S;
```

In both cases, if *Vn* is a field of both *V1* and *V2*, it's interpreted as *V2.Vn*, not *V1.Vn*.

If the selection of a record variable involves indexing an array or dereferencing a pointer, these actions are executed once before the component statement is executed.

Exception statements

The exceptions statements are the **raise** statement, the **try...except** statement, and the **try...finally** statement. These statements are described in Chapter 11, "Exceptions."

Chapter

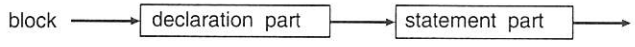
8

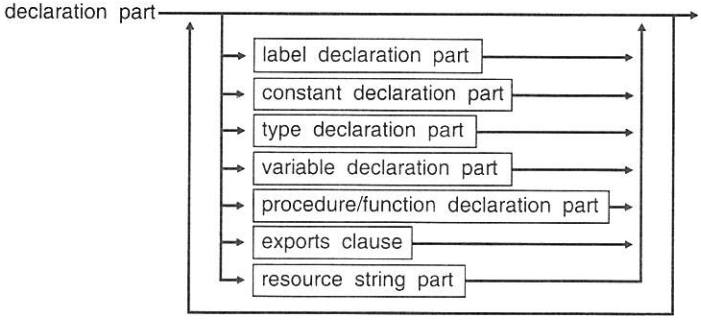
Blocks, locality, and scope

A *block* is made up of declarations, which are written and combined in any order, and statements. Each block is part of a procedure declaration, a function declaration, a method declaration, or a program or unit. All identifiers and labels declared in the declaration part are local to the block.

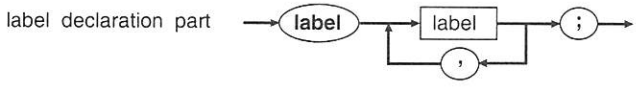
Blocks

The overall syntax of any block follows this format:



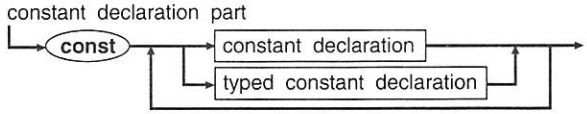


Labels that mark statements in the corresponding statement part are declared in the *label declaration part*. Each label must mark only one statement.

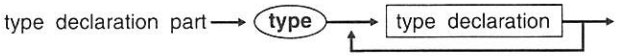


A label must be an identifier or a digit sequence in the range 0 to 9999.

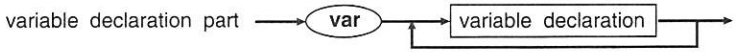
The *constant declaration part* consists of constant declarations local to the block.



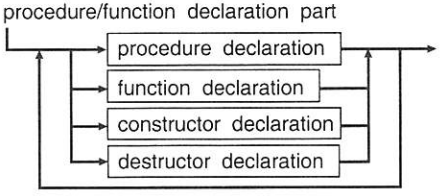
The *type declaration part* includes all type declarations local to the block.



The *variable declaration part* is composed of variable declarations local to the block.

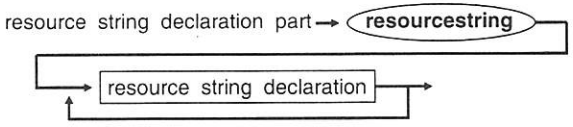


The procedure and function declarations local to the block make up the *procedure and function declaration part*.



The **exports** clause lists all procedures and functions that are exported by the current program or dynamic-link library. An **exports** clause is allowed only in the outermost declaration part of a program or dynamic-link library—it isn't allowed in the declaration part of a procedure, function, or unit. See "The exports clause" on page 14-6.

The **resourcestring** part lists all resource strings that be resourced and linked into the executable, DLL, or package. See "Resource strings" on page 3-2.



The *statement part* defines the statements or algorithmic actions to be executed by the block.

Rules of scope

The presence of an identifier or label in a declaration defines the identifier or label. Each time the identifier or label occurs again, it must be within the *scope* of this declaration.

Block scope

The scope of an identifier or label declared in a label, constant, type, variable, procedure, or function declaration stretches from the point of declaration to the end of the current block, and includes all blocks enclosed by the current block.

An identifier or label declared in an outer block can be *redeclared* in an inner block enclosed by the outer block. Before the point of declaration in the inner block, and after the end of the inner block, the identifier or label represents the entity declared in the outer block.

```
program Outer;          { Start of outer scope }
type
  I = Integer;          { define I as type Integer }
var
  T: I;                 { define T as an Integer variable }

procedure Inner;        { Start of inner scope }
type
  T = I;                { redefine T as type Integer }
var
  I: T;                 { redefine I as an Integer variable }
begin
  I := 1;
end;                    { End of inner scope }

begin
  T := 1;
end.                    { End of outer scope }
```

Record scope

The scope of a field identifier declared in a record-type definition extends from the point of declaration to the end of the record-type definition. Also, the scope of field identifiers includes field designators and **with** statements that operate on variable references of the given record type. See "Record types" on page 4-13.

Class scope

The scope of a component identifier declared in a class type extends from the point of declaration to the end of the class-type definition, and extends over all descendants of the class type and the blocks of all method declarations of the class type. Also, the scope of component identifiers includes field, method, and property designators, and

with statements that operate on variables of the given class type. For more information on classes, see Chapter 10, "Class types."

Unit scope

The scope of identifiers declared in the interface section of a unit follows the rules of block scope and extends over all *clients* of the unit. In other words, programs or units containing **uses** clauses have access to the identifiers belonging to the interface parts of the units in those **uses** clauses.

Each unit in a **uses** clause imposes a new scope that encloses the remaining units used and the program or unit containing the **uses** clause. The first unit in a **uses** clause represents the outermost scope, and the last unit represents the innermost scope. This implies that if two or more units declare the same identifier, an unqualified reference to the identifier selects the instance declared by the last unit in the **uses** clause. If you use a qualified identifier (a unit identifier, followed by a period, followed by the identifier), every instance of the identifier can be selected.

The identifiers of Object Pascal's predefined constants, types, variables, procedures, and functions act as if they were declared in a block enclosing all used units and the entire program. In fact, these standard objects are defined in a unit called *System*, which is used by any program or unit before the units named in the **uses** clause. This means that any unit or program can redeclare the standard identifiers, but a specific reference can still be made through a qualified identifier, for example, *System.Integer* or *System.Writeln*.

Chapter

9

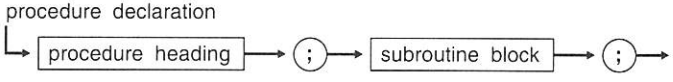
Procedures and functions

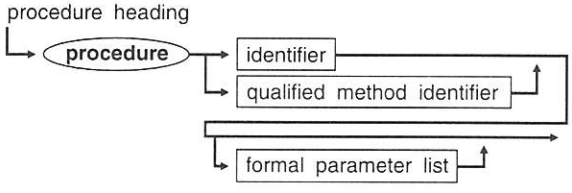
Procedures and functions let you nest additional blocks in the main program block. Each procedure or function declaration has a heading followed by a block. See Chapter 8, "Blocks, locality, and scope," for a definition of a block. A procedure is activated by a procedure statement; a function is activated by the evaluation of an expression that contains its call and returns a value to that expression.

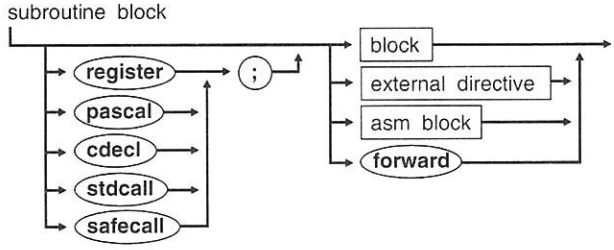
This chapter discusses the different types of procedure and function declarations and their parameters.

Procedure declarations

A *procedure declaration* associates an identifier with a block as a procedure; that procedure can then be activated by a procedure statement.







The procedure heading names the procedure's identifier and specifies the formal parameters (if any). The syntax for a formal parameter list is shown in "Parameters" on page 9-6.

A procedure is activated by a procedure statement, which states the procedure's identifier and actual parameters, if any. The statements to be executed on activation are noted in the statement part of the procedure's block. If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure is executed recursively (it calls itself while executing).

Here's an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(N mod 10 + Ord('0')) + S;
    N := N div 10;
  until N = 0;
  if N < 0 then S := '-' + S;
end;
```

Calling conventions

Object Pascal provides four directives that allow you to specify the calling conventions to be used for passing parameters to procedures and functions. The default calling convention is always **register**. Calling conventions differ in three areas:

- Order of passing the parameters
- Responsibility for removing parameters from the stack ("cleanup")
- Use of registers for passing parameters

The **register** and **pascal** conventions pass parameters from left to right, that is the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl**, **stdcall** and **safecall** conventions pass parameters from right to left. For all conventions except **cdecl**, the procedure or function removes parameters from the stack upon returning. With the **cdecl**

convention, the caller must remove parameters from the stack when the call returns. The **register** convention uses up to three CPU registers to pass parameters, whereas the other conventions always pass all parameters on the stack. The calling conventions are summarized in Table 9.1.

Table 9.1 Summary of calling conventions

Directive	Order	Cleanup	Registers
register	Left-to-right	Function	Yes
pascal	Left-to-right	Function	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Function	No
safecall	Right-to-left	Function	No

The **register** convention is by far the most efficient, since it often avoids the creation of a stack frame. The **pascal** and **cdecl** conventions are mostly useful for calling routines in dynamic-link libraries written in C, C++, or other languages. The **stdcall and safecall** conventions are used for calling Windows API routines. The **safecall** convention must be used to implement dual interfaces. For furthur information , see "Dual interfaces" on page 13-16.

For details on how each calling convention operates, see "Calling conventions" on page 19-1.

Forward declarations

A procedure or function declaration that specifies the directive **forward** instead of a block is a **forward** declaration. Somewhere after this declaration, the procedure must be defined by a *defining declaration*. The defining declaration can omit the formal parameter list and the function result, or it can optionally repeat it. In the latter case, the defining declaration's heading must match exactly the order, types, and names of parameters, and the type of the function result, if any.

No **forward** declarations are allowed in the **interface** part of a unit.

The **forward** declaration and the defining declaration must appear in the same procedure and function declaration part. Other procedures and functions can be declared between them, and they can call the forward-declared procedure. Therefore, mutual recursion is possible.

The **forward** declaration and the defining declaration constitute a complete procedure or function declaration. The procedure or function is considered declared at the **forward** declaration.

This is an example of a **forward** declaration:

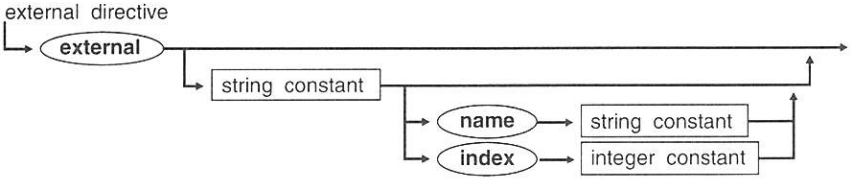
```
procedure Walter(M, N: Integer); forward;

procedure Clara(X, Y: Real);
begin
  :
  Walter(4, 5);
```

```
  :
end;

procedure Walter;
begin
  :
  Clara(8.3, 2.4);
  :
end;
```

A procedure's or function's defining declaration can be an **external** or **assembler** declaration; however, it can't be another **forward** declaration.

External declarations

With **external** declarations, you can interface with separately compiled procedures and functions written in assembly language. They also allow you to import procedures and functions from DLLs.



External directives consisting only of the reserved word **external** are used in conjunction with {**$L** filename} directives to link with **external** procedures and functions implemented in .OBJ files.

These are examples of **external** procedure declarations:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure MoveLong(var Source, Dest; Count: Integer); external;

procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
procedure FillLong(var Dest; Data: Longint; Count: Integer); external;

{$L BLOCK.OBJ}
```

External directives that specify a dynamic-link library name (and optionally an import name or an import ordinal number) are used to import procedures and functions from dynamic-link libraries. For example, the following **external** declaration imports a function called *MessageBox* from the DLL called "user32.dll" (part of the Windows API):

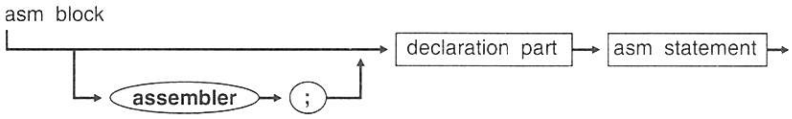
```
function MessageBox(HWnd: Integer; Text, Caption: PChar;
  Flags: Integer): Integer; stdcall;
  external 'user32.dll' name 'MessageBoxA';
```

To read more about importing procedures and functions from a DLL, see Chapter 14, "Dynamic-link libraries."

The **external** directive takes the place of the declaration and statement parts in an imported procedure or function, but aside from this difference in declaration, imported procedures and functions are just like regular procedures and functions.

Assembler declarations

With **assembler** declarations, you can write entire procedures and functions in inline assembly language.

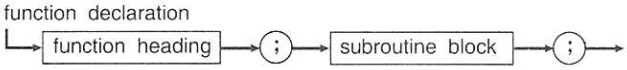


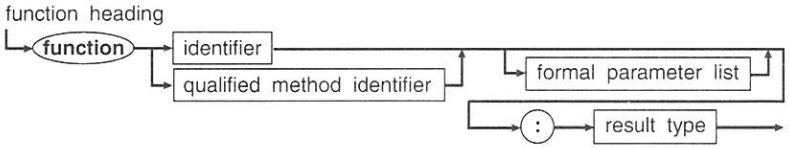
Note The optional assembler directive has no effect on the compiler and is accepted for backwards compatibility only.

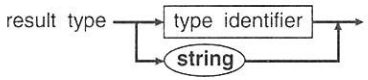
For more details on assembler procedures and functions, see online Help.

Function declarations

A function declaration defines a block that computes and returns a value.







The function heading specifies the identifier for the function, the formal parameters (if any), and the function result type.

A function is activated by the evaluation of a function call. The function call gives the function's identifier and actual parameters, if any, required by the function. A function call appears as an operand in an expression. When the expression is evaluated, the function is executed, and the value of the operand becomes the value returned by the function.

The statement part of the function's block specifies the statements to be executed upon activation of the function. The block should contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no such assignment statement exists or if it isn't executed, the value returned by the function is undefined.

If the function's identifier is used in a function call within the function's block, the function is executed recursively.

Every function implicitly has a local variable *Result* of the same type as the function's return value. Assigning to *Result* has the same effect as assigning to the name of the function. In addition, however, you can refer to *Result* in an expression, which refers to the current return value rather than generating a recursive function call.

Functions can return values of any type except file types.

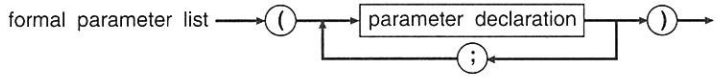
Following are examples of function declarations:

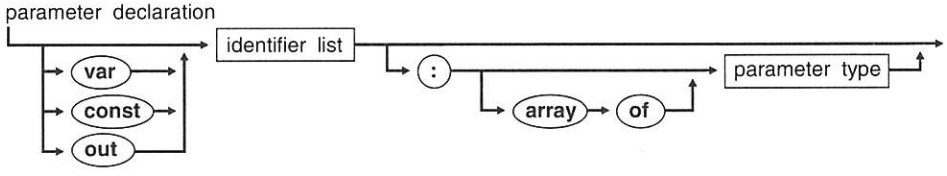
```
function Max(A: Vector; N: Integer): Extended;
var
  X: Extended;
  I: Integer;
begin
  X := A[1];
  for I := 2 to N do
    if X < A[I] then X := A[I];
  Max := X;
end;

function Power(X: Extended; Y: Integer): Extended;
var
  I: Integer;
begin
  Result := 1.0; I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Like procedures, functions can be declared as **forward**, **external**, or **assembler**.

Parameters

The declaration of a procedure or function specifies a *formal parameter list*. Each parameter declared in a formal parameter list is local to the procedure or function being declared and can be referred to by its identifier in the block associated with the procedure or function.





There are five kinds of parameters: *value, constant, variable, out* and *untyped*. These are characterized as follows:

- A parameter group without a preceding **var**, **out**, or **const** and followed by a type is a list of value parameters.
- A parameter group preceded by **const** and followed by a type is a list of constant parameters.
- A parameter group preceded by **var** and followed by a type is a list of variable parameters.
- A parameter group preceded by **out** and followed by a type is a list of out parameters.
- A parameter group preceded by **var**, **out**, or **const** and *not* followed by a type is a list of untyped parameters.

Array type parameters can be *open parameters*. Open parameters are described on "Open-array parameters" on page 9-9. A value, constant, or variable parameter declared using the syntax **array of** *T* is an *open-array parameter*. A value, constant, or variable parameter declared using the syntax **array of const** is a *type variant open-array parameter*.

Value parameters

A formal value parameter acts like a variable local to the procedure or function, except it gets its initial value from the corresponding actual parameter upon activation of the procedure or function. Changes made to a formal value parameter don't affect the value of the actual parameter.

A value parameter's corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of file type or of any structured type that contains a file type.

The actual parameter must be assignment-compatible with the type of the formal value parameter.

Constant parameters

A formal constant parameter acts like a local read-only variable that gets its value from the corresponding actual parameter upon activation of the procedure or function. Assignments to a formal constant parameter are *not* allowed, and likewise a formal constant parameter *can't* be passed as an actual variable parameter to another procedure or function.

A constant parameter's corresponding actual parameter in a procedure statement or function must follow the same rules as an actual value parameter.

In cases where a formal parameter never changes its value during the execution of a procedure or function, a constant parameter should be used instead of a value parameter. Constant parameters allow the implementor of a procedure or function to protect against accidental assignments to a formal parameter. Also, for structured-

and string-type parameters, the compiler can generate more efficient code when constant parameters are used instead of value parameters.

Variable parameters

A variable parameter is used when a value must be passed from a procedure or function to the caller. The corresponding actual parameter in a procedure statement or function call must be a variable reference. The formal variable parameter represents the actual variable during the activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected in the actual parameter.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (you can bypass this restriction through untyped parameters).

Note File types can be passed only as variable parameters.

If referencing an actual variable parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Out parameters

An out parameter is used when the called procedure or function parameter will exclusively return output data. Out parameters are resource managed; an out parameter reference is freed automatically by the compiler when the procedure or function is called. When a method with an out parameter is called, it simply fills the parameter with new information and on failure will set the parameter to **nil**.

Note It is recommended that you only use out parameters with COM methods since variable parameters are more efficient than out parameters. For information on COM, see Chapter 13, "Interfaces." In all other respects an out parameter behaves like a variable parameter.

Untyped parameters

When a formal parameter is an untyped parameter, the corresponding actual parameter can be any variable or constant reference, regardless of its type. An untyped parameter declared using the **var** or **out** keyword can be modified, whereas an untyped parameter declared using the **const** keyword is read-only.

Within the procedure or function, the untyped parameter is typeless; that is, it is incompatible with variables of all other types, unless it is given a specific type through a variable typecast.

This is an example of untyped parameters:

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
```

```
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

This function can be used to compare any two variables of any size. For example, given these declarations,

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

the following function calls

```
Equal(Vec1, Vec2, SizeOf(TVector))
Equal(Vec1, Vec2, SizeOf(Integer) * N)
Equal(Vec[1], Vec1[6], SizeOf(Integer) * 5)
Equal(Vec1[1], P, 4)
```

compare *Vec1* to *Vec2*, the first *N* components of *Vec1* to the first *N* components of *Vec2*, the first five components of *Vec1* to the last five components of *Vec1*, and *Vec1[1]* to *P.X* and *Vec1[2]* to *P.Y*.

While untyped parameters give you greater flexibility, they can be riskier to use. The compiler can't verify that operations on untyped variables are valid.

Open-array parameters

Open-array parameters allow arrays of varying sizes to be passed to the same procedure or function. A formal parameter declared using the syntax

```
array of T
```

is an *open-array parameter*. *T* must be a type identifier, and the actual parameter must be a variable of type *T*, or an array variable whose element type is *T*. Within the procedure or function, the formal parameter behaves as if it was declared as

```
array[0..N - 1] of T
```

where *N* is the number of elements in the actual parameter. In effect, the index range of the actual parameter is mapped onto the integers 0 to $N - 1$. If the actual parameter is a simple variable of type *T*, it is treated as an array with one element of type *T*.

A formal open-array parameter can be accessed by element only. Assignments to an entire open array aren't allowed, and an open array can be passed to other

procedures and functions only as an open-array parameter or as an untyped variable parameter.

Open-array parameters can be value, constant, and variable parameters and have the same semantics as regular value, constant, and variable parameters. In particular, assignments to elements of a formal open-array constant parameter are not allowed, and assignments to elements of a formal open-array value parameter don't affect the actual parameter.

Note For an open array value parameter, the compiler creates a local copy of the actual parameter within the procedure or function's stack frame. Therefore, be careful not to overflow the stack when passing large arrays as open array value parameters.

When applied to an open-array parameter, the *Low* standard function returns zero, the *High* standard function returns the index of the last element in the actual array parameter, and the *SizeOf* function returns the size of the actual array parameter.

The *Clear* procedure in the next example assigns zero to each element of an array of *Double*, and the *Sum* function computes the sum of all elements in an array of *Double*. Because the *A* parameter in both cases is an open-array parameter, the subroutines can operate on any array with an element type of *Double*.

```
procedure Clear(var A: array of Double);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end;

function Sum(const A: array of Double): Double;
var
  I: Integer;
  S: Double;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

When the element type of an open-array parameter is *Char*, the actual parameter may be a string constant. For example, given the procedure declaration,

```
procedure PrintStr(const S: array of Char);
var
  I: Integer;
begin
  for I := 0 to High(S) do
    if S[I] <> #0 then Write(S[I]) else Break;
end;
```

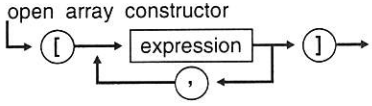
the following are valid procedure statements:

```
PrintStr('Hello world');
PrintStr('A');
```

When passed as an open-character array, an empty string is converted to a string with one element containing a NULL character, so the statement *PrintStr('')* is identical to the statement *PrintStr(#0)*.

Open-array constructors

Open-array constructors allow open-array parameters to be constructed directly within procedure and function calls. When a formal parameter of a procedure or function is an open-array value parameter or an open-array constant parameter, the corresponding actual parameter in a procedure or function call can be an open-array constructor.



An open-array constructor consists of one or more expressions separated by commas and enclosed in square brackets. Each expression must be assignment compatible with the element type of the open-array parameter. The use of an open-array constructor corresponds to creating a temporary array variable, and initializing the elements of the temporary array with the values given by the list of expressions. For example, given the declaration of the Sum function above, the statement

```
X := Sum([A, 3.14159, B + C]);
```

corresponds to

```
Temp[0] := A;
Temp[1] := 3.14159;
Temp[2] := B + C;
X := Sum(Temp);
```

where *Temp* is a temporary array variable with three elements of type *Double*.

Type variant open-array parameters

A type variant open-array parameter allows an open array of expressions of varying types to be passed to a procedure or function. A type variant open-array parameter is declared using the syntax

```
array of const
```

The **array of const** syntax is analogous to **array of** *TVarRec*. The *TVarRec* type is a variant record type which can represent values of integer, Boolean, character, real, string, pointer, class, class reference, and variant types. The *TVarRec* type is declared in the *System* unit as follows

```
type
  TVarRec = record
    case VType: Byte of
      vtInteger:    (VInteger: Integer);
      vtBoolean:    (VBoolean: Boolean);
```

```
    vtChar:       (VChar: Char);
    vtExtended:   (VExtended: PExtended);
    vtString:     (VString: PShortString);
    vtPointer:    (VPointer: Pointer);
    vtPChar:      (VPChar: PChar);
    vtObject:     (VObject: TObject);
    vtClass:      (VClass: TClass);
    vtWideChar:   (VWideChar: WideChar);
    vtPWideChar:  (VPWideChar: PWideChar);
    vtAnsiString: (VAnsiString: Pointer);
    vtCurrency:   (VCurrency: PCurrency);
    vtVariant:    (VVariant: PVariant);
end;
```

The *VType* field determines which value field is currently defined. Notice that for *vtExtended*, *vtString*, *vtCurrency*, and *vtVariant*, the corresponding fields contain pointers to the values rather than the values themselves. Also notice that for a long string value, the *VAnsiString* field is declared as *Pointer* because the long string type is not allowed in a variant record. When accessing the *VAnsiString* field, it must first be typecast to **string**. The *vtXXXX* value type constants are also declared in the *System* unit:

```
const
  vtInteger  = 0;
  vtBoolean  = 1;
  vtChar     = 2;
  vtExtended = 3;
  vtString   = 4;
  vtPointer  = 5;
  vtPChar    = 6;
  vtObject   = 7;
  vtClass    = 8;
  vtWideChar   = 9;
  vtPWideChar  = 10;
  vtAnsiString = 11;
  vtCurrency   = 12;
  vtVariant    = 13;
```

The *MakeStr* function below takes a type variant open-array parameter and returns a string that is the concatenation of the string representations of the arguments. The *IntToStr*, *FloatToStr*, and *CurrToStr* functions used by *MakeStr* are defined in the *SysUtils* unit.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
```

```
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
      end;
end;
```

When a formal parameter of a procedure or function is a type variant open-array value parameter or a type variant open-array constant parameter, the corresponding actual parameter in a procedure or function call can be an open-array constructor. The use of an open-array constructor for a type variant open-array parameter creates a temporary **array of** *TVarRec* and initializes the elements according to the type and value of each expression listed in the open-array constructor. For example, given the above function *MakeStr*, the statement

```
S := MakeStr(['Test ', 100, '-', True, '-', 3.14159]);
```

produces the following string

```
'Test 100-T-3.14159'
```

Table 9.2 lists the possible expression types in a type variant open-array constructor, and the corresponding value type codes.

Table 9.2 Type variant open-array expressions

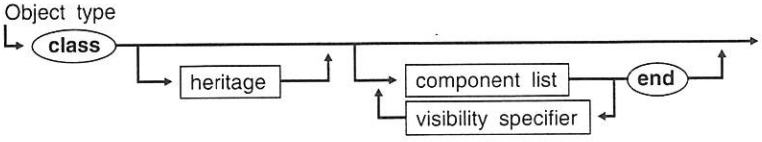
Type code	Expression type
vtInteger	Any integer type
vtBoolean	Any Boolean type
vtChar	Any character type
vtExtended	Any real type except *Currency*
vtString	Any short string type
vtPointer	Any pointer type except *PChar*
vtPChar	*PChar* or **array**[0..X] **of** *Char*
vtObject	Any class type
vtClass	Any class reference type
vtWideChar	Any wide character type
vtPWideChar	*PWideChar* or **array**[0..X] of *WideChar*
vtAnsiString	Long string (**string**)
vtCurrency	*Currency*
vtVariant	*Variant*

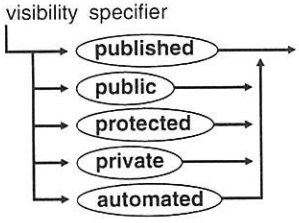
Chapter

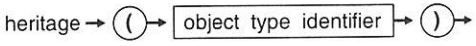
10

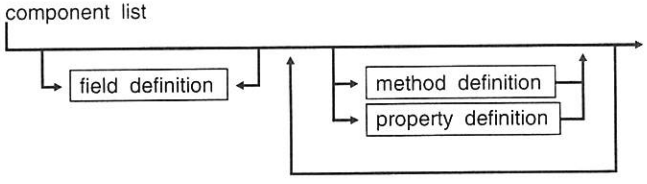
Class types

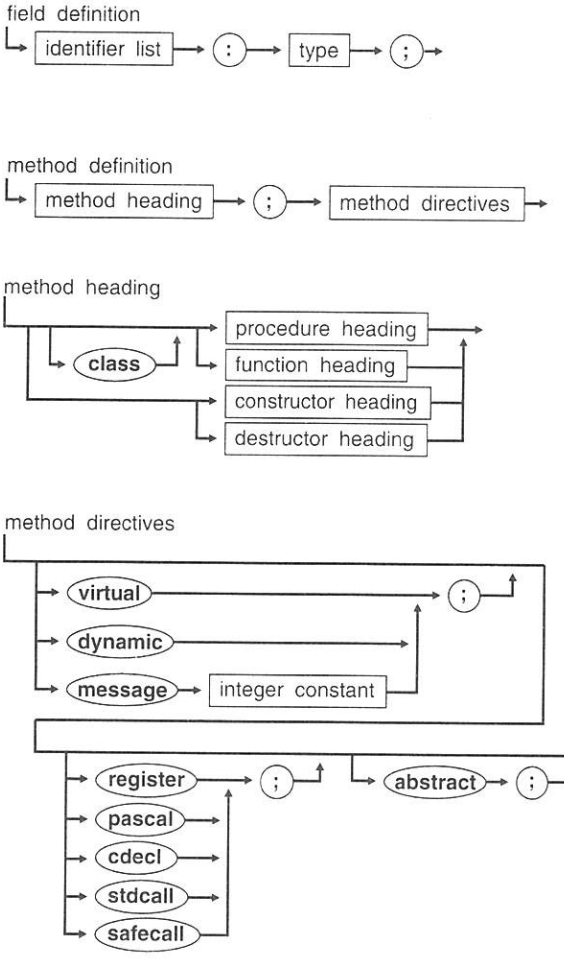
A class type is a structure consisting of a fixed number of components. The possible components of a class are *fields*, *methods*, and *properties*. Unlike other types, a class type can be declared only in a type declaration part in the outermost scope of a program or unit. Therefore, a class type can't be declared in a variable declaration part or within a procedure, function, or method block.











Instances and references

An *instance* of a class type is a dynamically allocated block of memory with a layout defined by the class type. Instances of a class type are also commonly referred to as *objects*. Objects are created using constructors, and destroyed using destructors. Each object of a class type has a unique copy of the fields declared in the class type, but all share the same methods.

A variable of a class type contains a *reference* to an object of the class type. The variable doesn't contain the object itself, but rather is a pointer to the memory block that has been allocated for the object. Like pointer variables, multiple class type variables can refer to the same object. Furthermore, a class type variable can contain the value **nil**, indicating that it doesn't currently reference an object.

Note Unlike a pointer variable, it isn't necessary to de-reference a class type variable to gain access to the referenced object. In other words, while it is necessary to write *Ptr^.Field* to access a field in a dynamically allocated record, the ^ operator is implied when accessing a component of an object, and the syntax is simply *Instance.Field*.

Throughout this book, the term *object reference* is used to denote a value of a class type. For example, a class-type variable contains an object reference, and a constructor returns an object reference.

In addition, the term *class reference* is used to denote a value of a class-reference type. For example, a class-type identifier is a class reference, and a variable of a class-reference type contains a class reference. Class-reference types are described in further detail on page 10-25.

Class components

The components of a class are fields, methods, and properties. Class components are sometimes referred to as members.

Fields

A field declaration in a class defines a data item that exists in each instance of the class. This is similar to a field of a record.

Methods

A method is a procedure or function that performs an operation on an object. Part of the call to a method specifies the object the method should operate on.

The declaration of a method within a class type corresponds to a **forward** declaration of that method. This means that somewhere after the class-type declaration, and within the same module, the method must be implemented by a defining declaration.

Within the implementation of a method, the identifier *Self* represents an implicit parameter that references the object for which the method was invoked.

Constructors and *destructors* are special methods that control construction and destruction of objects.

A constructor defines the actions associated with creating an object. When invoked, a constructor acts as a function that returns a reference to a newly allocated and initialized instance of the class type.

A destructor defines the actions associated with destroying an object. When invoked, a destructor will deallocate the memory that was allocated for the object.

A *class method* is a procedure or function that operates on a class reference instead of an object reference.

Properties

A property declaration in a class defines a named attribute for objects of the class and the actions associated with reading and writing the attribute. Properties are described in depth in "Property syntax" on page 10-18.

Inheritance

A class type can inherit components from another class type. If *T2* inherits from *T1*, then *T2* is a *descendant* of *T1*, and *T1* is an *ancestor* of *T2*. Inheritance is transitive; that is, if *T3* inherits from *T2*, and *T2* inherits from *T1*, then *T3* also inherits from *T1*. The *domain* of a class type consists of itself and all of its descendants.

A descendant class implicitly contains all the components defined by its ancestor classes. A descendant class can add new components to those it inherits. However, it can't remove the definition of a component defined in an ancestor class.

The predefined class type *TObject* is the ultimate ancestor of all class types. If the declaration of a class type doesn't specify an ancestor type (that is, if the heritage part of the class declaration is omitted), the class type will be derived from *TObject*. *TObject* is declared by the *System* unit, and defines a number of methods that apply to all classes. For a description of these methods, see "The TObject and TClass types" on page 10-28.

Components and scope

The scope of a component identifier declared in a class type extends from the point of declaration to the end of the class-type definition, and extends over all descendants of the class type and the blocks of all method declarations of the class type. Also, the scope of component identifiers includes field, method, and property designators, and **with** statements that operate on variables of the given class type.

A component identifier declared in a class type can be *redeclared* in the block of a method declaration of the class type. In that case, the *Self* parameter can be used to access the component whose identifier was redeclared.

A component identifier declared in an ancestor class type can be redeclared in a descendant of the class type. Such redeclaration effectively hides the inherited component, although the **inherited** keyword can be used to bring the inherited component back into scope.

Forward references

The declaration of a class type can specify the reserved word **class** and nothing else, in which case the declaration is a forward declaration. A forward declaration must be

resolved by a normal declaration of the class within the same type declaration part. Forward declarations allow mutually dependent classes to be declared. For example:

```
type
  TFigure = class;
  TDrawing = class
    Figure: TFigure;
      ⋮
  end;
  TFigure = class
    Drawing: TDrawing;
      ⋮
  end;
```

Class type compatibility rules

A class type is assignment-compatible with any ancestor class type; therefore, during program execution, a class-type variable can reference an instance of that type or an instance of any descendant type. For example, given the declarations

```
type
  TFigure = class
    ⋮
  end;
  TRectangle = class(TFigure)
    ⋮
  end;
  TRoundRect = class(TRectangle)
    ⋮
  end;
  TEllipse = class(TFigure)
    ⋮
  end;
```

a value of type *TRectangle* can be assigned to variables of type *TRectangle*, *TFigure*, and *TObject*, and during execution of a program, a variable of type *TFigure* might be either **nil** or reference an instance of *TFigure*, *TRectangle*, *TRoundRect*, *TEllipse*, or any other instance of a descendant of *TFigure*.

Component visibility

The visibility of a component identifier is governed by the visibility attribute of the component section that declares the identifier. The five possible visibility attributes are **published**, **public**, **protected**, **private**, and **automated**.

Component identifiers declared in the component list that immediately follows the class type heading have the published visibility attribute if the class type is compiled in the {**$M+**} state or is derived from a class that was compiled in the {**$M+**} state. Otherwise, such component identifiers have the public visibility attribute.

Public components

Component identifiers declared in **public** sections have no special restrictions on their visibility.

Published components

The visibility rules for published components are identical to those of public components. The only difference between published and public components is that *runtime type information* is generated for fields and properties that are declared in a **published** section. This runtime type information enables an application to dynamically query the fields and properties of an otherwise unknown class type.

Note The Delphi Visual Component Library uses runtime type information to access the values of a component's properties when saving and loading form files. Also, the Delphi development environment uses a component's runtime type information to determine the list of properties shown in the Object Inspector.

A class type cannot have **published** sections unless it is compiled in the {**$M+**} state or is derived from a class that was compiled in the {**$M+**} state. The **$M** compiler directive controls the generation of runtime type information for a class. For further details on **$M**, see the *User's Guide*.

Fields defined in a **published** section must be of a class type. Fields of all other types are restricted to **public**, **protected**, and **private** sections.

Properties defined in a **published** section cannot be array properties. Furthermore, the type of a property defined in a **published** section must be an ordinal type, a real type (*Single*, *Double*, *Extended*, *Comp*, or *Currency*, but not *Real*), a string type, a small set type, a class type, or a method pointer type. A small set type is a set type with a base type whose lower and upper bounds have ordinal values between 0 and 31. In other words, a small set type is a set that fits in a byte, a word, or a double word.

Automated components

The visibility rules for automated components are identical to those of public components. The only difference between automated and public components is that *automation type information* is generated for methods and properties that are declared in an automated section. This automation type information makes it possible to create *OLE Automation servers*.

Note Automated sections are typically only used in classes derived from the *TAutoObject* class defined in the *OleAuto* unit. *Automated* remains in the language only for backwards compatibility with Delphi 2.0. The *TAutoObject* from *ComObj.pas* doesn't use the **automated** directive to distinguish automated components. For further details on creating OLE Automation servers with Delphi, see the *User's Guide*.

The following restrictions apply to methods and properties declared in an automated section:

- For a method, the types of all method parameters and the function result (if any) must be automatable. Likewise, for a property, the property type and the types of any array property parameters must be automatable. The automatable types are: *Byte*, *Currency*, *Double*, *Integer*, *Single*, *Smallint*, **string**, *TDateTime*, *Variant*, and *WordBool*. Declaring methods or properties that use non-automatable types in an **automated** section results in an error.
- Method declarations must use the **register** calling convention (the default).
- Methods can be **virtual**, but not **dynamic**.
- Property declarations can only include access specifiers (**read** and **write**). No other specifiers (**index**, **stored**, **default**, **nodefault**) are allowed.
- Property access specifiers must list a method identifier. Field identifiers are not allowed.
- Property access methods must use the register calling convention.
- Property overrides (property declarations that don't include the property type) are not allowed.

A method or property declaration in an **automated** section can include an optional **dispid** directive, which must be followed by an integer constant that gives the OLE Automation dispatch ID of the method or property. If a **dispid** directive is not present, the compiler automatically picks a number one larger than the largest dispatch ID used by any method or property in the class and its ancestors. Specifying an already used dispatch ID in a **dispid** directive causes an error.

Protected components

When accessing through a class type declared in the current module, the protected component identifiers of the class and its ancestors are visible. In all other cases, protected component identifiers are hidden.

Access to protected components of a class is restricted to the implementation of methods of the class and its descendants. Therefore, components of a class that are intended for use only in the implementation of derived classes are usually declared as protected.

Private components

The visibility of a component identifier declared in a **private** component section is restricted to the module that contains the class-type declaration. In other words, private component identifiers act like normal public component identifiers within the module that contains the class-type declaration, but outside the module, any private component identifiers are unknown and inaccessible. By placing related class types in the same module, these class types can gain access to each other's private components without making the private components known to other modules.

Static methods

Methods declared in a class type are by default *static*. When a static method is called, the *declared* (compile-time) type of the class or object used in the method call determines which method implementation to activate. In the following example, the *Draw* methods are static:

```
type
  TFigure = class
    procedure Draw;
    :
  end;
  TRectangle = class(TFigure)
    procedure Draw;
    :
  end;
```

The following section of code illustrates the effect of calling a static method. Even though in the second call to *Figure.Draw* the *Figure* variable references an object of class *TRectangle*, the call invokes the implementation of *TFigure.Draw* because the declared type of the *Figure* variable is *TFigure*.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;{ Invokes TFigure.Draw }
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;{ Invokes TFigure.Draw }
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;{ Invokes TRectangle.Draw }
  Rectangle.Destroy;
end;
```

Virtual methods

A method can be made *virtual* by including a **virtual** directive in its declaration. When a virtual method is called, the *actual* (runtime) type of the class or object used in the method call determines which method implementation to activate. The following is an example of a declaration of a virtual method:

```
type
  TFigure = class
    procedure Draw; virtual;
    :
  end;
```

A virtual method can be *overridden* in a descendant class. When an **override** directive is included in the declaration of a method, the method overrides the inherited

implementation of the method. An override of a virtual method must match exactly the order and types of the parameters, and the function result type (if any), of the original method.

The *only* way a virtual method can be overridden is through the **override** directive. If a method declaration in a descendant class specifies the same method identifier as an inherited method, but doesn't specify an **override** directive, the new method declaration will *hide* the inherited declaration, but not override it.

Assuming the declaration of class *TFigure* above, the following two descendant classes override the *Draw* method:

```
type
  TRectangle = class(TFigure)
    procedure Draw; override;
    :
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
    :
  end;
```

The following section of code illustrates the effect of calling a virtual method through a class-type variable whose actual type varies at runtime:

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;{ Invokes TRectangle.Draw }
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;{ Invokes TEllipse.Draw }
  Figure.Destroy;
end;
```

Dynamic methods

A method is made dynamic by including a **dynamic** directive in its declaration. Dynamic methods are semantically identical to virtual methods. Virtual and dynamic methods differ only in the implementation of method call dispatching at runtime; for all other purposes, the two types of methods can be considered equivalent.

In the implementation of virtual methods, the compiler favors speed of call dispatching over code size. The implementation of dynamic methods on the other hand favors code size over speed of call dispatching.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful only in situations where a base class declares a large number of virtual methods, and an application declares a large number of descendant classes with few overrides of the inherited virtual methods.

Abstract methods

An *abstract* method is a virtual or dynamic method whose implementation isn't defined in the class declaration in which it appears; its definition is instead deferred to descendant classes. An abstract method in effect defines an interface, but not the underlying operation.

A method is abstract if an **abstract** directive is included in its declaration. A method can be declared **abstract** only if it is first declared **virtual** or **dynamic**. The following is an example of a declaration of an abstract method.

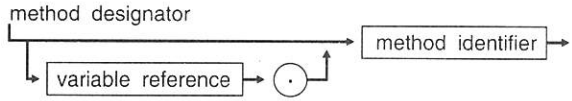
```
type
  TFigure = class
    procedure Draw; virtual; abstract;
    :
  end;
```

An override of an abstract method is identical to an override of a normal virtual or dynamic method, except that in the implementation of the overriding method, an **inherited** method isn't available to call.

Calling an abstract method through an object that hasn't overridden the method will generate an exception at runtime.

Method activations

A method is activated (or called) through a function call or procedure statement consisting of a *method designator* followed by an actual parameter list. This type of call is known as *method activation*.



The variable reference specified in a method designator must denote an object reference or a class reference, and the method identifier must denote a method of that class type.

The instance denoted by a method designator becomes an implicit actual parameter of the method; it corresponds to a formal parameter named *Self* that possesses the class type corresponding to the activated method.

Within a **with** statement that references an object or a class, the variable-reference part of a method designator can be omitted. In that case, the implicit *Self* parameter of the method activation becomes the instance referenced by the **with** statement.

Method implementations

The declaration of a method within a class type corresponds to a **forward** declaration of that method. Somewhere after the class-type declaration, and within the same

module, the method must be implemented by a defining declaration. For example, given the class-type declaration

```
type
  TFramedLabel = class(TLabel)
  protected
    procedure Paint; override;
  end;
```

the *Paint* method must later be implemented by a defining declaration. For example,

```
procedure TFramedLabel.Paint;
begin
  inherited Paint;
  with Canvas do
  begin
    Brush.Color := clWindowText;
    Brush.Style := bsSolid;
    FrameRect(ClientRect);
  end;
end;
```

For procedure and function methods, the defining declaration takes the form of a normal procedure or function declaration, except that the procedure or function identifier in the heading is a *qualified method identifier*. A qualified method identifier consists of a class-type identifier followed by a period (.) and then by a method identifier.

For constructors and destructors, the defining declaration takes the form of a procedure method declaration, except that the **procedure** reserved word is replaced by **constructor** or **destructor**.

A method's defining declaration can optionally repeat the formal parameter list of the method heading in the class type. The defining declaration's method heading must match exactly the order, types, and names of the parameters, and the type of the function result, if any.

In the defining declaration of a method, there is always an implicit parameter with the identifier *Self*, corresponding to a formal parameter of the class type. Within the method block, *Self* represents the instance for which the method was activated.

The scope of a component identifier in a class type extends over any procedure, function, constructor, or destructor block that implements a method of the class type. The effect is the same as if the entire method block was embedded in a **with** statement of the form:

```
with Self do
begin
  :
end;
```

Within a method block, the reserved word **inherited** can be used to access redeclared and overridden component identifiers. For example, in the implementation of the *TFramedLabel.Paint* method above, **inherited** is used to invoke the inherited implementation of the *Paint* method. When an identifier is prefixed with **inherited**,

the search for the identifier begins with the immediate ancestor of the enclosing method's class type.

Constructors and destructors

Constructors and destructors are special methods that control construction and destruction of objects. A class can have zero or more constructors and destructors for objects of the class type. Each is specified as a component of the class in the same way as a procedure or function method, except that the reserved words **constructor** and **destructor** begin each declaration instead of **procedure** and **function**. Like other methods, constructors and destructors can be inherited.

Constructors

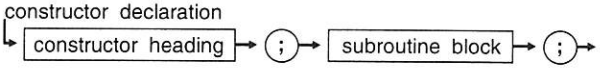
Constructors are used to create and initialize new objects. Typically, the initialization is based on values passed as parameters to the constructor.

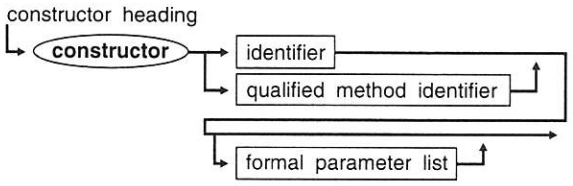
Unlike an ordinary method, which must be invoked on an object reference, a constructor can be invoked on either a class reference or an object reference.

To create a new object, a constructor must be invoked on a class reference. When a constructor is invoked on a class reference, the following actions take place:

- Storage for a new object is allocated from the heap.
- The allocated storage is cleared. This causes the ordinal value of all ordinal type fields to become zero, the value of all pointer and class-type fields to become **nil**, and the value of all string fields to become empty.
- The user-specified actions of the constructor are executed.
- A reference to the newly allocated and initialized object is returned from the constructor. The type of the returned value is the same as the class type specified in the constructor call.

When a constructor is invoked on an object reference, the constructor acts like a normal procedure method. This means that a new object is *not* allocated and cleared, and that the constructor call does *not* return an object reference. Instead, the constructor operates on the specified object reference, and only executes the user specified actions given in the constructor's statement part. A constructor is typically invoked on an object reference only in conjunction with the **inherited** keyword to execute an inherited constructor.





An example of a class type and its associated constructor follows below:

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    :
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);{ Initialize inherited parts }
  Width := 65;{ Change inherited properties }
  Height := 65;
  FPen := TPen.Create;{ Initialize new fields }
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is almost always to call an inherited constructor to initialize the inherited fields of the object. Following that, the constructor then initializes the fields of the object that were introduced in the class. Because a constructor always clears the storage it allocates for a new object, all fields automatically have a default value of zero (ordinal types), **nil** (pointer and class types), empty (string types), or *Unassigned* (the *Variant* type). Unless a field's default value is non-zero, there is no need to initialize the field in a constructor.

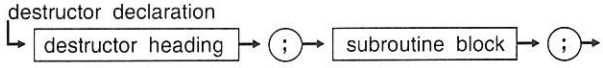
If an exception occurs during execution of a constructor that was invoked on a class reference, the *Destroy* destructor is automatically called to destroy the unfinished object. The effect is the same as if the entire statement part of the constructor were embedded in a **try...finally** statement of this form:

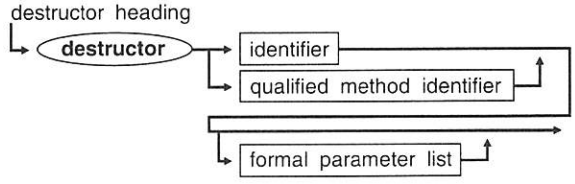
```
try
  :{ User defined actions }
except{ On any exception }
  Destroy;{ Destroy unfinished object }
  raise;{ Re-raise exception }
end;
```

Like other methods, constructors can be virtual. When invoked through a class-type identifier, as is usually the case, a virtual constructor is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects—that is, construction of objects whose types aren't known at compile time, as described in "Constructors and class references" on page 10-27.

Destructors

Destructors are used to destroy objects. When a destructor is invoked, the user-defined actions of the destructor are executed, and then the storage that was allocated for the object is disposed of. The user-defined actions of a destructor typically consist of destroying any embedded objects and releasing any resources that were allocated by the object.





The following example shows how the destructor that was declared for the *TShape* class in the preceding section might be implemented.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

The last action of a destructor is typically to call the inherited destructor to destroy the inherited fields of the object.

While it is possible to declare multiple destructors for a class, it is recommended that classes only implement overrides of the inherited *Destroy* destructor. *Destroy* is a parameterless virtual destructor declared in *TObject*, and because *TObject* is the ultimate ancestor of every class, the *Destroy* destructor always available for any object.

As described in the preceding section on constructors, if an exception occurs during the execution of a constructor, the *Destroy* destructor is invoked to destroy the unfinished object. This means that destructors must be prepared to handle destruction of *partially constructed* objects. Because a constructor sets all fields of a new object to null values before executing any user-defined actions, any class-type or pointer-type fields in a partially constructed object are always **nil**. A destructor

should therefore always check for **nil** values before performing operations on class-type or pointer-type fields.

Referring to the *TShape.Destroy* destructor mentioned earlier, note that the *Free* method (which is inherited from *TObject*) is used to destroy the objects referenced by the *FPen* and *FBrush* fields. The implementation of the *Free* method is

```
procedure TObject.Free;
begin
  if Self <> nil then Destroy;
end;
```

The *Free* method is a convenient way of checking for **nil** before invoking *Destroy* on an object reference. By calling *Free* instead of *Destroy* for any class-type fields, a destructor is automatically prepared to handle partially constructed objects resulting from constructor exceptions. For that same reason, direct calls to *Destroy* aren't recommended.

Class operators

Object Pascal defines two operators, **is** and **as**, that operate on class and object references.

The is operator

The **is** operator is used to perform *dynamic type checking*. Using the **is** operator, you can check whether the *actual* (runtime) type of an object reference belongs to a particular class. The syntax of the **is** operator is

```
ObjectRef is ClassRef
```

where *ObjectRef* is an object reference and *ClassRef* is a class reference. The **is** operator returns a Boolean value. The result is *True* if *ObjectRef* is an instance of the class denoted by *ClassRef* or an instance of a class derived from the class denoted by *ClassRef*. Otherwise, the result is *False*. If *ObjectRef* is **nil**, the result is always *False*. If the declared types of *ObjectRef* and *ClassRef* are known not to be related—that is, if the declared type of *ObjectRef* is known not to be an ancestor of, equal to, or a descendant of *ClassRef*—the compiler reports a type-mismatch error.

The **is** operator is often used in conjunction with an **if** statement to perform a *guarded typecast*. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

Here, if the **is** test is *True*, it is safe to typecast *ActiveControl* to be of class *TEdit*.

The rules of operator precedence group the **is** operator with the relational operators (=, <>, <, >, <=, >=, and **in**). This means that when combined with other Boolean expressions using the **and** and **or** operators, **is** tests must be enclosed in parentheses:

```
if (Sender is TButton) and (TButton(Sender).Tag <> 0) then ...;
```

The as operator

The **as** operator is used to perform *checked typecasts*. The syntax of the **as** operator is

```
ObjectRef as ClassRef
```

where *ObjectRef* is an object reference and *ClassRef* is a class reference. The resulting value is a reference to the same object as *ObjectRef*, but with the type given by *ClassRef*. When evaluated at runtime, *ObjectRef* must be **nil**, an instance of the class denoted by *ClassRef*, or an instance of a class derived from the class denoted by *ClassRef*. If none of these conditions are *True*, an exception is raised. If the declared types of *ObjectRef* and *ClassRef* are known not to be related—that is, if the declared type of *ObjectRef* is known not to be an ancestor of, equal to, or a descendant of *ClassRef*—the compiler reports a type-mismatch error.

The **as** operator is often used in conjunction with a **with** statement. For example,

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

The rules of operator precedence group the **as** operator with the multiplying operators (*, /, **div**, **mod**, **and**, **shl**, and **shr**). This means that when used in a variable reference, an **as** typecast must be enclosed in parentheses:

```
(Sender as TButton).Caption := '&Ok';
```

Message handling

Message handler methods are used to implement user-defined responses to *dynamically dispatched messages*. Delphi's Visual Class Library uses message handler methods to implement Windows message handling.

Message handler declarations

A message handler method is defined by including a **message** directive in the method declaration.

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ...
  end;
```

A message handler method must be a procedure that takes a single variable parameter, and the **message** keyword must be followed by an integer constant between 1 and 49151 which specifies the *message ID*.

Note When declaring a message handler method for a VCL control, the integer constant specified in the message directive must be a Windows message ID. The *Messages* unit defines all Windows message IDs and their corresponding message records.

In contrast to a regular method, a message handler method does not have to specify an **override** directive in order to override an inherited message handler. In fact, a message handler override doesn't even have to specify the same method identifier and parameter name and type as the method it overrides. The message ID solely determines which message the method will respond to, and whether or not it is an override.

Message handler implementations

The implementation of a message handler method corresponds to that of a normal method. For example, the *TTextBox.WMChar* message handler method defined above might be implemented as follows

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

The implementation of a message handler method can call the inherited implementation using an **inherited** statement as shown above. The **inherited** statement automatically passes the message record as a parameter to the inherited method. The **inherited** statement invokes the first message handler with the same message ID found in the most derived ancestor class (that is, the ancestor class that is closest to the class in the inheritance hierarchy). If none of the ancestor classes implement a message handler for the given message ID, the **inherited** statement instead calls the *DefaultHandler* virtual method, which is inherited from *TObject* and therefore present in any class.

The effect of the **inherited** statement is that for a particular message ID, a class does not need to know whether its parent classes implement a handler for the message–an inherited implementation always appears to be available.

Message dispatching

Message handler methods are typically not called directly. Instead, messages are dispatched to an object using the *Dispatch* method defined in the *TObject* class. *Dispatch* is declared as

```
procedure TObject.Dispatch(var Message);
```

The *Message* parameter passed to *Dispatch* must be a record, and the first entry in the record must be a field of type *Cardinal* which contains the message ID of the message being dispatched. For example

```
type
```

```
TMessage = record
  Msg: Cardinal;
  ...
end;
```

A message record can contain any number of additional fields that define message specific information.

A call to *Dispatch* invokes the most derived implementation of a message handler for the given message ID. In other words, *Dispatch* invokes the first message handler with a matching message ID found by examining the class itself, its ancestor, its ancestor's ancestor, and so on until *TObject* is reached. If the class and its ancestors do not define a handler for the given message ID, *Dispatch* will instead invoke the *DefaultHandler* method.

The *DefaultHandler* method is declared in *TObject* as follows

```
procedure DefaultHandler(var Message); virtual;
```

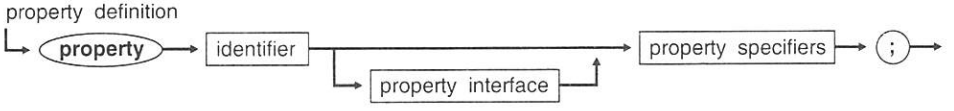
The implementation of *DefaultHandler* in *TObject* simply returns without performing any actions. By overriding *DefaultHandler*, a class can implement default handling of messages. As described above, *DefaultHandler* is invoked when *Dispatch* is called to dispatch a message for which the class implements no message handler. *DefaultHandler* is also invoked when a message handler method executes an **inherited** statement for which no inherited message handler exists.

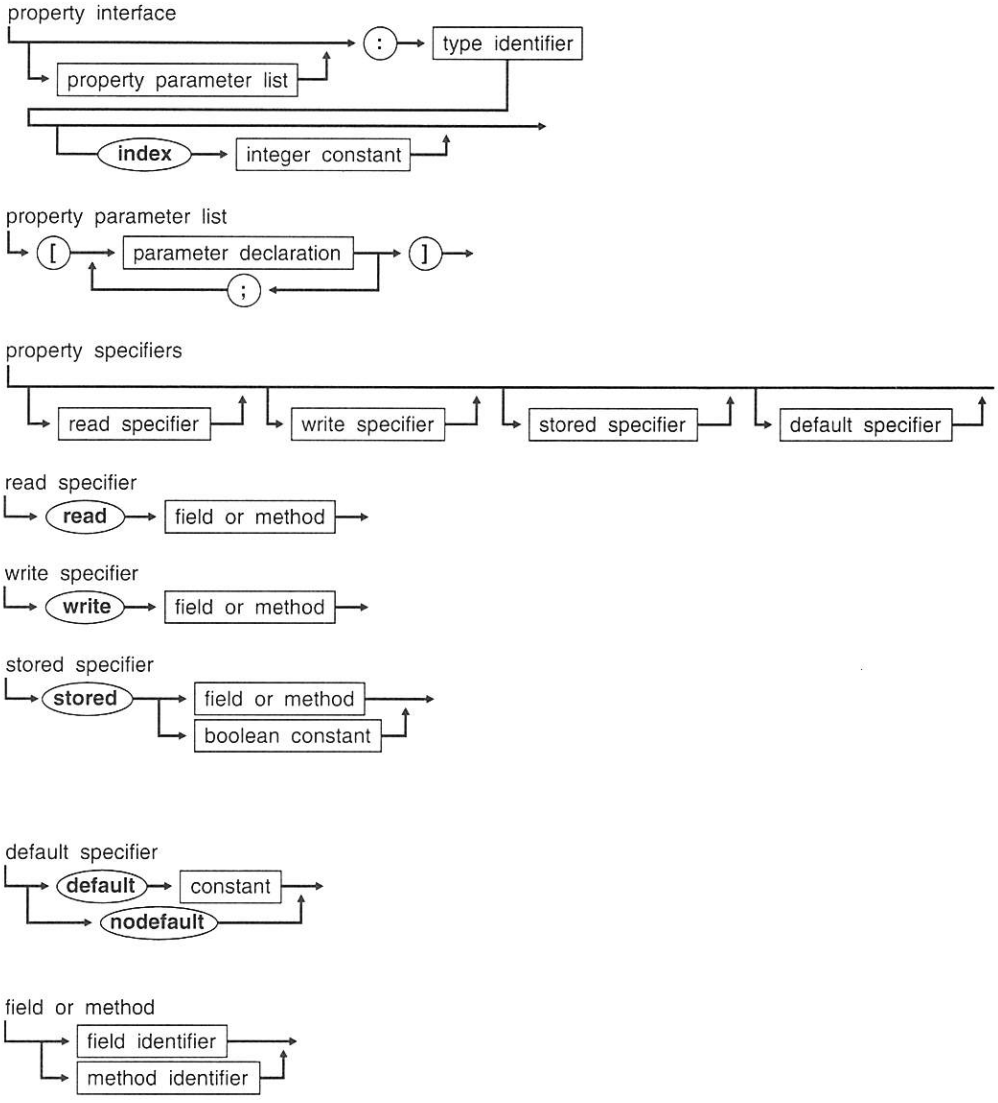
Note The *DefaultHandler* method for a VCL control invokes the *DefWindowProc* default message handling function defined by Windows.

Property syntax

A property definition in a class declares a named attribute for objects of the class and the actions associated with reading and writing the attribute. Examples of properties are the caption of a form, the size of a font, the name of a database table, and so on.

Properties are a natural extension of fields in an object. Both can be used to express attributes of an object, but whereas fields are merely storage locations which can be examined and modified at will, properties provide greater control over *access* to attributes, they provide a mechanism for associating *actions* with the reading and writing of attributes, and they allow attributes to be *computed*.





Property definitions

The definition of a property specifies the name and type of the property, and the actions associated with reading (examining) and writing (modifying) the property. A property can be of any type except a file type.

The declarations below define an imaginary *TCompass* control which has a *Heading* property that can assume values from 0 to 359 degrees. The definition of the *Heading*

property further states that its value is read from the *FHeading* field, and that its value is written using the *SetHeading* method.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    :
  end;
```

Property access

When a property is referenced in an expression, its value is read using the field or method listed in the **read** specifier, and when a property is referenced in an assignment statement, its value is written using the field or method listed in the **write** specifier. For example, assuming that *Compass* is an object reference of the *TCompass* type defined above, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the *TCompass* class, no action is associated with reading the *Heading* property. The read operation simply consists of examining the value stored in the *FHeading* field. On the other hand, assigning a value to the *Heading* property translates into a call to the *SetHeading* method, which not only stores the new value in the *FHeading* field, but also performs whatever actions are required to update the user interface of the compass control. For example, *SetHeading* might be implemented as this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;
  end;
end;
```

Note Unlike fields, properties can't be passed as variable parameters, and it isn't possible to take the address of a property using the @ operator. This is true even if the **read** and **write** specifiers both list a field identifier, thereby ensuring that a future implementation of the property is free to change one or both access specifiers to list a method.

Access specifiers

The **read** and **write** specifiers of a property definition control how a property is accessed. A property definition must include at least a **read** or a **write** specifier, but isn't required to include both. If a property definition includes only a **read** specifier, then the property is said to be *read-only* property, and if a property definition includes only a **write** specifier, then the property is said to be *write-only property*. If a property definition includes both a **read** and a **write** specifier, the property is said to be *read-write* property. It is an error to assign a value to a read-only property. Likewise, it is an error to use a write-only property in an expression.

The **read** or **write** keyword in an access specifier must be followed by a field identifier or a method identifier. The field or method can belong to the class type in which the property is defined, in which case the field or method definition must precede the property definition, or it can belong to an ancestor class, in which case the field or method must be visible in the class containing the property definition.

Fields and methods listed in access specifiers are governed by the following rules:

- If an access specifier lists a field identifier, then the field type must be identical to the property type.
- If a **read** specifier lists a method identifier, then the method must be a parameterless function method, and the function result type must be identical to the property type.
- If a **write** specifier lists a method identifier, then the method must be a procedure method that takes a single value or constant parameter of the same type as the property type.

For example, if a property is defined as

```
property Color: TColor read GetColor write SetColor;
```

then preceding the property definition, the *GetColor* method must be defined as

```
function GetColor: TColor;
```

and the *SetColor* method must be defined as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

Array properties

Array properties allow the implementation of *indexed* properties. Examples of array properties include the items of a list, the child controls of a control, and the pixels of a bitmap.

The definition of an array property includes an *index parameter list* which specifies the names and types of the indexes of the array property. For example,

```
property Objects[Index: Integer]: TObject
  read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor
```

```
    read GetPixel write SetPixel;
  property Values[const Name: string]: string
    read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure or function's formal parameter list, except that the parameter declarations are enclosed in square brackets instead of parentheses. Note that unlike array types, which can only specify ordinal type indexes, array properties allow indexes of any type. For example, the *Values* property declared previously might represent a lookup table in which a string index is used to look up a string value.

An access specifier of an array property must list a method identifier. In other words, the **read** and **write** specifiers of an array property aren't allowed to specify a field name. The methods listed in array property access specifiers are governed by the following rules:

- The method listed in the **read** specifier of an array property must be a function that takes the same number and types of parameters as listed in the property's index parameter list, and the function result type must be identical to the property type.
- The method listed in the **write** specifier of an array property must be a procedure with the same number and types of parameters as listed in the property's index parameter list, plus an additional value or constant parameter of the same type as the property type.

For example, for the array properties declared above, the access methods might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;

procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by following the property identifier with a list of actual parameters enclosed in square brackets. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

The definition of an array property can be followed by a **default** directive, in which case the array property becomes the *default array property* of the class. For example,

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    :
  end;
```

If a class has a default array property, then access to the array property of the form *Instance.Property[...]* can be abbreviated to *Instance[...]*. For example, given that *StringArray* is an object reference of the *TStringArray* class defined above, the construct

```
StringArray.Strings[Index]
```

can be abbreviated to

```
StringArray[Index]
```

When an object reference is followed by a list of indexes enclosed in square brackets, the compiler automatically selects the class type's default array property, or issues an error if the class type has no default array property.

If a class defines a default array property, derived classes automatically inherit the default array property. It isn't possible for a derived class to redeclare or hide the default array property.

Index specifiers

The definition of a property can optionally include an *index specifier*. Index specifiers allow a number of properties to share the same access methods. An index specifier consists of the directive **index** followed by an integer constant with a value between –32767 and 32767. For example,

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
    property Left: Longint index 0
      read GetCoordinate write SetCoordinate;
    property Top: Longint index 1
      read GetCoordinate write SetCoordinate;
    property Right: Longint index 2
      read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3
      read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint
      read GetCoordinate write SetCoordinate;
      .
      .
  end;
```

An access specifier of a property with an **index** specifier must list a method identifier. In other words, the **read** and **write** specifiers of a property with an **index** specifier aren't allowed to list a field name.

When accessing a property with an **index** specifier, the integer value specified in the property definition is automatically passed to the access method as an extra parameter. For that reason, an access method for a property with an **index** specifier must take an extra value parameter of type *Integer*. For a property read function, the

extra parameter must be the last parameter. For a property write procedure, the extra parameter must be the second to last parameter, that is it must immediately precede the parameter that specifies the new property value.

Assuming that *Rectangle* is an object reference of the *TRectangle* type defined above, the statement

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage specifiers

The optional **stored**, **default**, and **nodefault** specifiers of a property definition are called *storage specifiers*. They control certain aspects of the runtime type information that is generated for **published** properties. Storage specifiers are only supported for normal (non-array) properties.

Storage specifiers have no semantic effects on a property, that is they do not affect how a property is used in program code. The Delphi Visual Component Library, however, uses the information generated by storage specifiers to control *filing* of a component's properties—the automatic saving and loading of a component's property values in a form file. The **stored** directive controls whether a property is filed, and the **default** and **nodefault** properties control the value that is considered a property's default value.

If present in a property definition, the **stored** keyword must be followed by a Boolean constant (*True* or *False*), the identifier of a field of type *Boolean*, or the identifier of a parameterless function method with returns a value of type *Boolean*. If a property definition doesn't include a **stored** specifier, the results are the same as if a **stored** *True* specifier were included.

The **default** and **nodefault** specifiers are supported only for properties of ordinal types and small set types. If present in a property definition, the **default** keyword must be followed by a constant of the same type as the property. If a property definition doesn't (or can't) include a **default** or **nodefault** specifier, the results are the same as if a **nodefault** specifier were included.

When saving a component's state, the Delphi Visual Component Library iterates over all of the component's **published** properties. For each property, the result of evaluating the Boolean constant, field, or function method of the **stored** specifier controls whether the property is saved. If the result is *False*, the property isn't saved. If the result is *True*, the property's current value is compared to the value given in the **default** specifier (if present). If the current value is equal to the default value, the property isn't saved. Otherwise, if current value is different from the default value, or if the property has no default value, the property is saved.

Property overrides

A property definition that doesn't include a property interface is called a *property override*. A property override allows a derived class to change the visibility, access specifiers, and storage specifiers of an inherited property.

In its simplest form, a property override specifies only the reserved word **property** followed by an inherited property identifier. This form is used to change the visibility of a property. If, for example, a base class defines a property in a **protected** section, a derived class can raise the visibility of the property by declaring a property override in a **public** or **published** section.

A property override can include a **read**, **write**, **stored**, and **default** or **nodefault** specifier. Any such specifier overrides the corresponding inherited specifier. Note that a property override can change an inherited access specifier or add a missing access specifier, but it can't remove an access specifier.

The following declarations illustrate the use of property overrides to change the visibility, access specifiers, and storage specifiers of inherited properties:

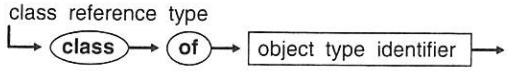
```
type
  TBase = class
    :
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    :
  end;

type
  TDerived = class(TBase)
    :
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    :
  end;
```

The property override of *Size* adds a **write** specifier to allow the *Size* property to be modified. The property overrides of *Text* and *Color* change the visibility of the properties from protected to published. In addition, the property override of *Color* specifies that the property should be filed if its value isn't *clBlue*.

Class-reference types

Class-reference types allow operations to be performed directly on classes. This contrasts with class types, which allow operations to be performed on instances of

classes. Class-reference types are sometimes referred to as *metaclasses* or *metaclass types*.



Class-reference types are useful in the following situations:

- With a virtual constructor to create an object whose actual type is unknown at compile time
- With a class method to perform an operation on a class whose actual type is unknown at compile time
- As the right operand of an **is** operator to perform a dynamic type check with a type that is unknown at compile time
- As the right operand of an **as** operator to perform a checked typecast to a type that is unknown at compile time

The declaration of a class-reference type consists of the reserved words **class of** followed by a class-type identifier. For example,

```
type
  TComponent = class(TPersistent)
    :
  end;
  TComponentClass = class of TComponent;
  TControl = class(TComponent)
    :
  end;
  TControlClass = class of TControl;

var
  ComponentClass: TComponentClass;
  ControlClass: TControlClass;
```

The previous declarations define *TComponentClass* as a type that can reference class *TComponent*, or any class that derives from *TComponent*, and *TControlClass* as a type that can reference class *TControl*, or any class that derives from *TControl*.

Class-type identifiers function as *values* of their corresponding class-reference types. For example, in addition to its other uses, the *TComponent* identifier functions as a value of type *TComponentClass*, and the *TControl* identifier functions as a value of type *TControlClass*.

A class-reference type value is assignment-compatible with any ancestor class-reference type. Therefore, during program execution, a class-reference type variable can reference the class it was defined for or any descendant class of the class it was defined for. Referring to the previous declarations, the assignments

```
ComponentClass := TComponent;      { Valid }
ComponentClass := TControl;        { Valid }
```

are both valid. Of these assignments,

```
ControlClass := TComponent;        { Invalid }
ControlClass := TControl;          { Valid }
```

only the second one is valid, however. The first assignment is an error because *TComponent* isn't a descandant of *TControl*, and therefore not a value of type *TControlClass*.

A class-reference type variable can be **nil**, which indicates that the variable doesn't currently reference a class.

Every class inherits (from *TObject*) a method function called *ClassType*, which returns a reference to the class of an object. The type of the value returned by *ClassType* is *TClass*, which is declared as **class of** *TObject*. This means that the value returned by *ClassType* may have to be typecast to a more specific descendant type before it can be used. For example:

```
if Control <> nil then
  ControlClass := TControlClass(Control.ClassType) else
  ControlClass := nil;
```

Constructors and class references

A constructor can be invoked on a variable reference of a class-reference type. This allows *polymorphic construction* of objects; that is, construction of objects whose actual type isn't known at compile time. For example,

```
function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

The *CreateControl* function uses a class-reference type parameter to specify which class of control to create. It subsequently uses the class-reference type parameter to invoke the *Create* constructor of the class. Because class-type identifiers also function as class-reference type values, calls to *CreateControl* can simply specify the identifier of the class to create an instance for. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
CreateControl(TButton, 'Button1', 120, 10, 80, 30);
```

Constructors that are invoked through class-reference types are usually virtual. That way, the constructor implementation that is called depends on the *actual* (runtime) class type selected by the class reference.

Class methods

A class method is a method that operates on a class reference instead of an object reference. The definition of a class method must include the reserved word **class** before the **procedure** or **function** keyword that starts the definition. For example,

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    :
  end;
```

The defining declaration of a class method must also start with the reserved word **class**. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  :
end;
```

In the defining declaration of a class method, the identifier *Self* represents the class for which the method was activated. The type of *Self* in a class method is **class of** *ClassType*, where *ClassType* is the class type for which the method is implemented. Because *Self* doesn't represent an object reference in a class method, it isn't possible to use *Self* to access fields, properties, and normal methods. It is possible, however, to call constructors and other class methods through *Self*.

A class method can be invoked through a class reference or an object reference. When invoked through an object reference, the class of the given object reference is passed as the *Self* parameter.

The TObject and TClass types

The *System* unit defines two types, *TObject* and *TClass*, which serve as the *root* types for all class types and class-reference types. The declarations of the two types are shown here.

```
type
  TObject = class;
  TClass = class of TObject;
  TObject = class
    constructor Create;
    destructor Destroy; virtual;
    class function ClassInfo: Pointer;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    function ClassType: TClass;
    procedure CleanupInstance;
    procedure DefaultHandler(var Message); virtual;
```

```
  procedure Dispatch(var Message);
  function FieldAddress(const Name: ShortString): Pointer;
  procedure Free;
  procedure FreeInstance; virtual;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function InitInstance(Instance: Pointer): TObject;
  class function InstanceSize: Longint;
  class function NewInstance: TObject; virtual;
  class function MethodAddress(const Name: ShortString): Pointer;
  class function MethodName(Address: Pointer): ShortString;
end;
```

Chapter

11

Exceptions

An exception is generally an error condition or other event that interrupts normal flow of execution in an application. When an exception is *raised*, it causes control to be transferred from the current point of execution to an *exception handler*. Object Pascal's exception handling support provides a structured means of separating normal program logic from error handling logic, greatly increasing the maintainability and robustness of applications.

Object Pascal uses objects to represent exceptions. This has several advantages, the key ones of which are

- Exceptions can be grouped into hierarchies using inheritance
- New exceptions can be introduced without affecting existing code
- An exception object can carry information (such as an error message or an error code) from the point where it was raised to the point where it is handled

Using exception handling

The *SysUtils* unit implements exception generation and handling for the Object Pascal runtime library. When an application uses the *SysUtils* unit, all runtime errors are automatically converted into exceptions. This means that error conditions such as out of memory, division by zero, and general protection fault, which would otherwise terminate an application, can be caught and handled in a structured fashion.

Note Delphi's Visual Class Library fully supports exception handling. An application that uses VCL automatically also uses the *SysUtils* unit, thus enabling exception handling.

Exception declarations

An exception in Object Pascal is simply a class, and the declaration of an exception is no different than the declaration of an ordinary class. Although it is possible to use an instance of any class as an exception object, it is recommended that all exceptions be derived from the *Exception* class defined in the *SysUtils* unit. For example,

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

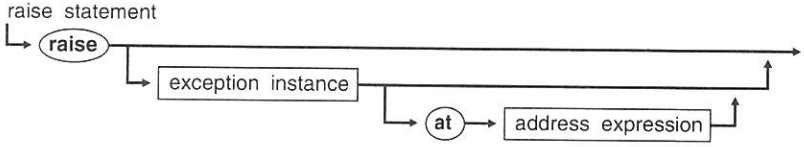
Exceptions are often grouped into families of related exceptions using inheritance. The above declarations (which were extracted from the *SysUtils* unit) define a family of related math error exceptions. Using families, it is possible to handle an entire group of exceptions under one name. For example, an exception handler for *EMathError* will also handle *EInvalidOp*, *EZeroDivide*, *EOverflow*, and *EUnderflow* exceptions, and any user-defined exceptions that directly or indirectly derive from *EMathError*.

Exception classes sometimes define additional fields, methods, and properties used to convey additional information about the exception. For example, the *EInOutError* class defined in the *SysUtils* unit introduces an *ErrorCode* field which contains the file I/O error code that caused the exception.

```
type
  EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

The raise statement

An exception is raised using a **raise** statement.



The argument to a **raise** statement must be an object. In other words, the **raise** keyword must be followed by an expression of a class type. When an exception is raised, the exception handling logic takes ownership of the exception object. Once the exception is handled, the exception object is automatically destroyed through a call to the object's *Destroy* destructor. An application should never attempt to manually destroy a raised exception object.

Note A **raise** statement raises an object, not a class. The argument to a **raise** statement is typically constructed "on the fly" through a call to the *Create* constructor of the appropriate exception class.

A **raise** statement that omits the exception object argument will *re-raise* the current exception. This form of a **raise** statement is allowed only in an *exception block*, and is described further in "Re-raising exceptions" on page 11-6.

Control *never* returns from a **raise** statement. Instead, **raise** transfers control to the innermost exception handler that can handle exceptions of the given class. Innermost in this case means the handler whose **try**...**except** block was most recently entered and not yet exited.

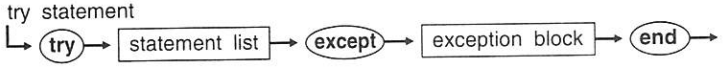
The *StrToIntRange* function below converts a string to an integer, and raises an *ERangeError* exception if the resulting value is not within a specified range.

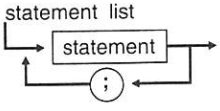
```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

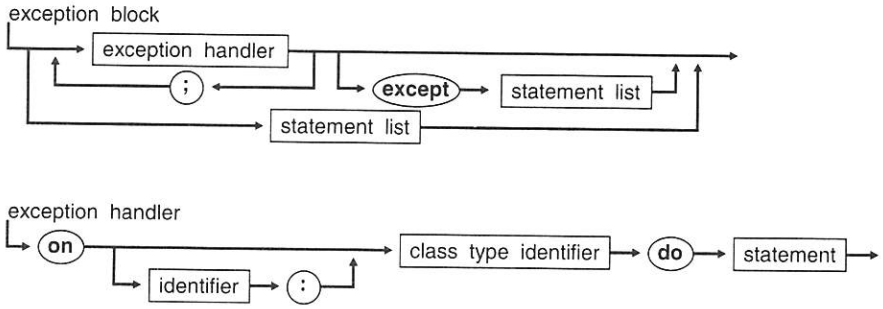
Note Raising an exception in the initialization code (see "The initialization part" on page 12-3) of a unit may not produce the desired behavior for the following reason. Exception support comes from the *SysUtils* unit, so a unit which uses exceptions must import *SysUtils*. With packages (see Chapter 14, "Packages"), the startup of a program has changed. An RTL helper function is called to initialize all the units, but if an exception occurs during initialization, all initialized units wll be finalized and the exception re-raised. In this example, *SysUtils* will get initialized, then another unit raises the exception and *SysUtils* gets finalized and the exception is re-raised. *System* will catch the exception and since no exeption handler is installed, attempts to handle it all by itself. It does not understand *SysUtils* based exceptions, so it maps it to something else that it finds reasonable, usually the *Ctrl-c* interrupt.

The try...except statement

Exceptions are handled using **try**...**except** statements.







A **try...except** statement executes the statements in the *try statement list* in sequential order. If the statements execute without any exceptions being raised, the *exception block* is ignored, and control is passed to the statement following the **end** keyword that ends the **try...except** statement.

The *exception block* in the **except...end** section defines exception handlers for the *try statement list*. An exception handler can be invoked only by a **raise** statement executed in the *try statement list* or by a procedure or function called from the *try statement list*.

When an exception is raised, control is transferred to the innermost exception handler that can handle exceptions of the given class. The search for an exception handler starts with the most recently entered and not yet exited **try...except** statement. If that **try...except** statement cannot handle exceptions of the given class, the next most recently entered **try...except** statement is examined. This *propagation* of the exception continues until an appropriate handler is found, or there are no more active **try...except** statements. In the latter case a runtime error occurs and the application is terminated.

To determine whether the *exception block* of a **try...except** statement can handle a particular exception, the **on...do** exception handlers are examined in order of appearance. The first exception handler that lists the exception class or a base class of the exception is considered a match. If an *exception block* contains an **else** part, and if none of the **on...do** exception handlers match the exception, the **else** part is considered a match. An *exception block* that contains only a statement list is considered a match for any exception.

Once a matching exception handler is found, the stack is "unwound" to the procedure or function that contains the handler, and control is transferred to the handler. The unwinding process will discard all procedure and function calls that occurred since entering the **try...except** statement containing the exception handler.

Following execution of an exception handler, the exception object is automatically destroyed through a call to the object's *Destroy* destructor, and control is passed to the statement following the **end** keyword that ends the **try...except** statement.

In the example below, the first **on...do** handles division by zero exceptions, the second **on...do** handles overflow exceptions, and the final **on...do** handles all other math exceptions.

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

As described earlier, to locate a handler for a given exception class, the **on...do** handlers are processed in order of appearance. This means that handlers for the most derived classes should be listed first. For example, since *EZeroDivide* and *EOverflow* are both derived from *EMathError*, a handler for *EMathError* will also handle *EZeroDivide* and *EOverflow* exceptions. If *EMathError* was listed before these exceptions, the more specific handlers would never be invoked.

An **on...do** exception handler can optionally specify an identifier and a colon before the exception class identifier. This declares the identifier to represent the exception object during execution of the statement that follows **on...do**. For example,

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

The scope of an identifier declared in an exception handler is the statement that follows **on...do**. The identifier hides any similarly named identifier in an outer scope.

If a **try...except** statement specifies an **else** part, then any exceptions that aren't handled by the **on...do** exception handlers will be handled by the **else** part. For example,

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** part will handle any exception that isn't an *EMathError*.

An *exception block* that contains no **on...do** handlers, but instead consists only of a list of statements, will handle all exceptions.

```
try
  ...
except
  HandleException;
end;
```

Here, any exception that occurs as a result of executing the statements between **try** and **except** will be handled by the *HandleException* procedure.

Re-raising exceptions

In some situations, a procedure or function may need to perform clean-up operations when an exception occurs, but the procedure or function may not be prepared to actually handle the exception. For example, consider the *GetFileList* function below, which allocates a *TStringList* object and fills it with the file names matching a given search path.

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
    begin
      Result.Add(SearchRec.Name);
      I := FindNext(SearchRec);
    end;
  except
    Result.Free;
    raise;
  end;
end;
```

The function first allocates a new *TStringList* object, and then uses the *FindFirst* and *FindNext* functions (defined in the *SysUtils* unit) to initialize the string list. If, for any reason, the initialization of the string list fails, for example because the given search path is invalid, or because there is not enough memory to fill in the string list, *GetFileList* needs to dispose the newly allocated string list, since the caller does not yet know of its existence. For that reason, the initialization of the string list is performed in a **try**...**except** statement. If an exception occurs, the exception handler part of the **try**...**except** statement disposes the string list, and then *re-raises* the exception.

Nested exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as exceptions raised in an exception handler are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the *Tan* function shown below.

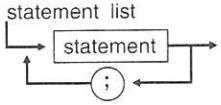
```
type
  ETrigError = class(EMathError);
function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
```

```
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
  end;
end;
```

If an *EMathError* is raised in the computation of the tangent of the given angle, such as the *EZeroDivide* that would result if *Cos*(*X*) is zero, the exception handler raises an *ETrigError* exception. Since the *Tan* function does not provide a handler for the *ETrigError* exception that it raises, the *ETrigError* will propagate beyond the original exception handler, causing the *EMathError* exception to be disposed. To the caller, the result is simply that the *Tan* function has raised an *ETrigError* exception.

The try...finally statement

When a section of code acquires a resource, it is often necessary to ensure that the resource be released again, regardless of whether the code completes normally or is interrupted by an exception. For example, a section of code that opens and processes a file will normally want to ensure that the file is closed no matter how the code terminates. The **try**...**finally** statement can be used in such situations.



A **try**...**finally** statement executes the statements in the *try statement list* in sequential order. If no exceptions are raised in the *try statement list*, the *finally statement list* is executed. If an exception is raised in the *try statement list*, control is transferred to the *finally statement list*, and once the *finally statement list* completes execution, the exception is re-raised. The resulting effect is that the *finally statement list* is always executed, regardless of how the *try statement list* terminates.

The section of code shown below illustrates how a **try**...**finally** statement can be used to ensure that a file that was opened is always closed.

```
Reset(F);
try
  ProcessFile(F);
finally
  CloseFile(F);
end;
```

Note In a typical application, **try...finally** statements such as the one above tend to be much more common than **try...except** statements. Applications written using Delphi's Visual Class Library, for example, normally rely on VCL's default exception handling mechanisms, thus seldom needing to use **try...except** statements. Resource allocations will however often have to be guarded against exceptions, and the use of **try...finally** statements will therefore be much more frequent.

If an exception is raised but not handled in the *finally statement list*, that exception is propagated out of the **try...finally** statement, and any original exception is lost.

Note It is strongly recommended that a *finally statement list* always handle all local exceptions, so as to not disturb the propagation of an external exception.

Exit, Break, and Continue procedures

If a call to one of the *Exit*, *Break*, or *Continue* standard procedures cause control to leave the *try statement list* of a **try...finally** statement, the *finally statement list* is automatically executed. Likewise, if one of these standard procedures are used to leave an exception handler, the exception object is automatically disposed.

Predefined exceptions

The *SysUtils* unit declares a number of exception classes, including a class named *Exception* which serves as the ultimate ancestor of all exception classes. The public interface of *Exception* is declared as follows

```
type
  Exception = class(TObject)
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Integer);
    constructor CreateResFmt(Ident: Integer; const Args: array of const);
    constructor CreateHelp(const Msg: string; HelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string; const Args: array of const;
      HelpContext: Integer);
    constructor CreateResHelp(Ident, HelpContext: Integer);
    constructor CreateResFmtHelp(Ident: Integer; const Args: array of const;
    HelpContext: Integer);
    property HelpContext: Integer;
    property Message: string;
  end;
```

The *Exception* class establishes two properties, *Message* and *HelpContext*, that every exception object inherits. This means that an arbitrary exception object can at least provide a descriptive message of the exception condition, and possibly a help context that refers to further online help on the topic.

The constructors defined by *Exception* provide various ways of initializing the *Message* and *HelpContext* properties. In general

- Constructors that don't include "Res" in their names require the exception message to be specified as a string parameter. Those that do include "Res" in their names will initialize the *Message* property from the string resource with the given ID.
- Constructors that include "Fmt" in their names interpret the specified exception message as a format string, and require an extra *Args* parameter which supplies

the format arguments. The initial value of the *Message* property is constructed through a call to the *Format* function in the *SysUtils* unit.

- Constructors that include "Help" in their names require an extra *HelpContext* parameter which supplies the online help context for the exception.

For further information on specific exceptions, see online help.

Exception handling support routines

The *SysUtils* unit defines a number of exception handling support routines. For further information on specific routines, see online help.

Chapter

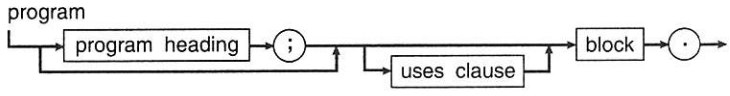
12

Programs and units

A program in Delphi is made up of individual source code modules called *units*. Units are the basis of modular programming in Object Pascal. They're used to create libraries you can include in various programs without making the source code available, and to divide large programs into logically related modules.

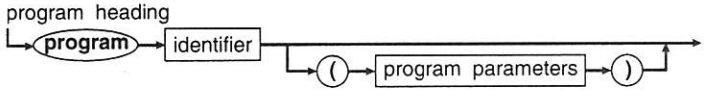
Program syntax

A Object Pascal program consists of a program heading, an optional **uses** clause, and a block.



The program heading

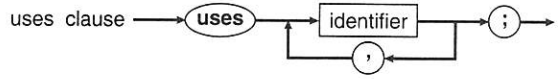
The program heading specifies the program's name and its parameters.



The program heading, if present, is ignored by the compiler.

The uses clause

The **uses** clause identifies all units used by the program.



The *System* unit is always used automatically and cannot be used explicitly by a unit. *System* implements all low-level, runtime routines to support such features as file input and output (I/O), string handling, floating point, dynamic memory allocation, and others.

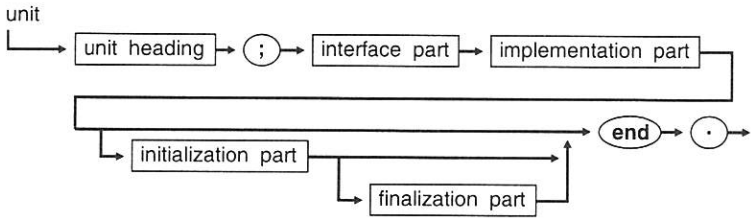
Apart from *System*, Object Pascal implements many standard units that aren't used automatically; you must include them in your **uses** clause. For example,

```
uses SysUtils;                    { Can now use SysUtils }
```

The order of the units listed in the **uses** clause determines the order of their initialization (see "The initialization part" on page 12-3) and affects the way that identifiers are located by the compiler. If two units declare a same-named type, the compiler will always use the one from the first unit listed in the **uses** clause. To access the identifier from the other unit, it must be fully qualified.

To find the unit file containing a compiled unit, the compiler adds the file extension .DCU to the unit name listed in the **uses** clause.

Unit syntax



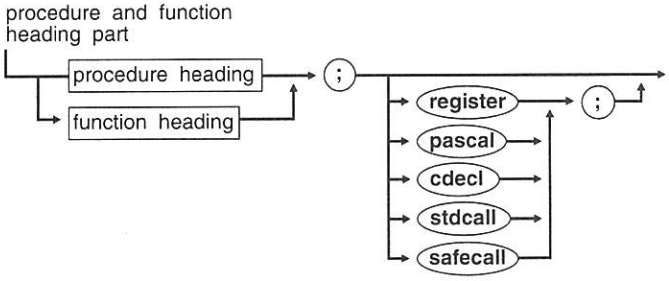
The unit heading

The unit heading specifies the unit's name.

The unit name is used when referring to the unit in a **uses** clause. The name must be unique: Two units with the same name can't be used at the same time.

The interface part

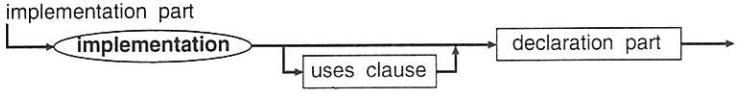
The interface part declares constants, types, variables, procedures, and functions that are *public*; that is, available to the host (the program or unit using the unit). The host can access these entities as if they were declared in a block that encloses the host.



Unless a procedure or function is **inline**, the interface part only lists the procedure or function heading. The block of the procedure or function follows in the implementation part.

The implementation part

The implementation part defines the block of all public procedures and functions. In addition, it declares constants, types, variables, procedures, and functions that are *private*; that is, not available to the host.

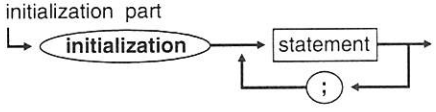


In effect, the procedure and function declarations in the interface part are like forward declarations, although the **forward** directive isn't specified. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation part.

Note Procedure and function headings can be duplicated from the interface part. You don't have to specify the formal parameter list, but if you do, the compiler will issue a compile-time error if the interface and implementation declarations don't match.

The initialization part

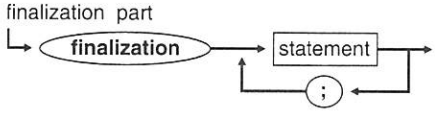
The optional initialization part consists of the reserved word **initialization**, followed by a list of statements which initialize the unit.



The initialization parts of units used by a program are executed in the same order that the units appear in the **uses** clause.

The finalization part

The finalization part is optional and can only appear if a unit also has an initialization part. The finalization part consists of the reserved word **finalization**, followed by a list of statements which finalize the unit. Finalization is the counterpart of initialization, and any resources (memory, files, etc.) acquired by a unit in its initialization part are typically released in the finalization part.



Unit finalization sections execute in the opposite order of initializations. For example, if your application initializes units *A*, *B*, and *C*, in that order, it will finalize them in the order *C*, *B*, and *A*.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if an exception is raised, the initialization code might not execute completely.

Indirect unit references

The **uses** clause in a module (program or unit) need only name the units used directly by that module. Consider the following:

```
program Prog;
uses Unit2;
const a = b;
begin
end.

unit Unit2;
interface
uses Unit1;
const b = c;
implementation
end.

unit Unit1;
interface
const c = 1;
implementation
const d = 2;
end.
```

In the previous example, *Unit2* is directly dependent on *Unit1* and *Prog* is directly dependent on *Unit2*. Also, *Prog* is indirectly dependent on *Unit1* (through *Unit2*), even though none of the identifiers declared in *Unit1* are available to *Prog*.

To compile a module, the compiler must be able to locate all units the module depends upon, directly or indirectly. So, to compile *Prog*, the compiler must be able to locate both *Unit1* and *Unit2*, or else an error occurs.

Note for C and other language users: The **uses** clauses of an Object Pascal program provides the "make" logic information traditionally found in make or project files of other languages. With the **uses** clause, Object Pascal can build all the dependency information into the module itself and reduce the chance for error.

When changes are made in the interface part of a unit, other units using the unit must be recompiled. If you use Make or Build, the compiler does this for you automatically. If changes are made only to the implementation or the initialization part, however, other units that use the unit *don't* have to be recompiled. In the previous example, if the interface part of *Unit1* is changed (for example, $c = 2$) *Unit2* must be recompiled; changing the implementation part (for example, $d = 1$) doesn't require recompilation of *Unit2*.

Object Pascal can tell when the interface part of a unit has changed or any compiler options that affect generated code have changed by computing a *unit version number* when the unit is compiled. In the preceding example, when *Unit2* is compiled, the current version number of *Unit1* is saved in the compiled version of *Unit2*. When *Prog* is compiled, the version number of *Unit1* is checked against the version number stored in *Unit2*. If the version numbers don't match (indicating that a change was made in the interface part of *Unit1* because *Unit2* was compiled or some compiler option has changed), the compiler reports an error or recompiles *Unit2*, depending on the mode of compilation.

Circular unit references

When two or more units reference each other in their **uses** clauses, the units are said to be mutually dependent. Any pattern of mutual dependencies is allowed as long as there are no circular references created by following the **uses** clause in the **interface** parts of the units. In other words, for any given set of mutually dependent units, starting with the **uses** clause of the **interface** part of any unit, it must not be possible to return to that unit by following references in the **uses** clauses of the **interface** parts. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the **uses** clause of the **implementation** part of at least one unit.

In the simplest case of two mutually dependent units, the implication of the above rule is that it is not possible for both units to reference the other in the **interface** part. It is however possible for one of the units to reference the other in the **interface** part, as long as the other references the first in the **implementation** part.

When the compiler detects an invalid circular reference path, it reports a "Circular unit reference" error. In order to reduce the chances of invalid circular references occurring, it is recommended that a unit use other units in the **implementation** part whenever possible. Only when symbols from another unit are needed in the **interface** part should that unit be listed in the **interface** part **uses** clause.

Chapter

13

Interfaces

Delphi 3.0 extends Object Pascal to add the concept of object interfaces as a native part of the language.

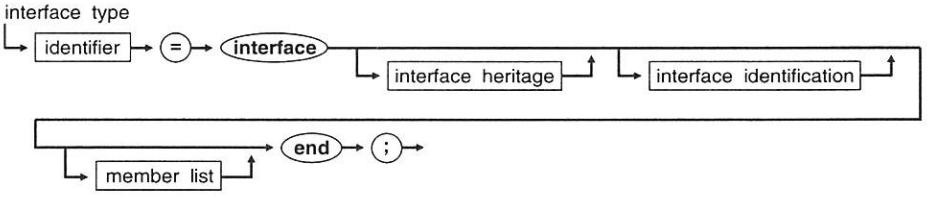
An object interface defines a set of methods that can be called on an object from any code outside that object's implementation. Like a unit interface, an object interface allows client code to access the object without needing to know anything about the object's implementation. The object's implementation class can remain hidden in the unit that implements it.

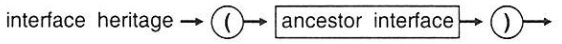
The declaration of an interface does not imply an actual implementation of the interface. Interfaces are never implemented by themselves, rather they are always implemented as part of a class.

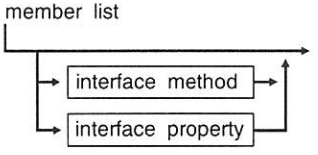
An interface reference is a reference to an implementation of an interface. Through an interface reference it is possible to invoke methods of a given interface without any knowledge of the actual implementation of the interface. Only methods declared in the interface may be called using the interface reference.

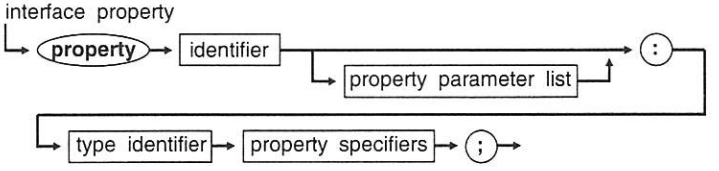
In Delphi, object interfaces are also compatible with the Component Object Model (COM) specification for interfaces. This means that any object built with Delphi that supports one or more interfaces is automatically a COM object. COM specifies a language-independent binary protocol for objects to interact with each other. The set of interfaces the object implements define a contract that allow COM objects to interact. Delphi objects with interfaces can interact with COM objects implemented in C++, Java, and other languages.

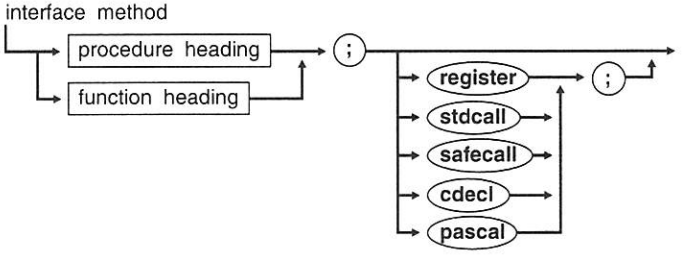
Interface syntax

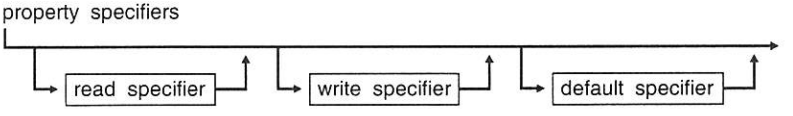


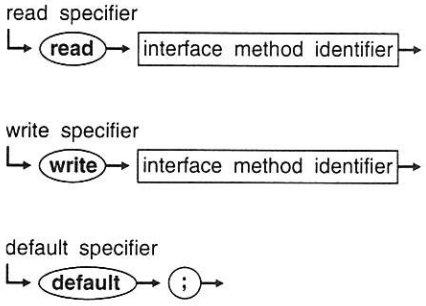












The following is an example of an interface declaration:

```
type
  IMalloc = interface(IUnknown)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

The following rules apply when declaring an interface:

- Interfaces can only be declared in a type declaration section in the outermost block of a module.
- An interface cannot be declared in a procedure or function.
- Like class types, an interface type can only be declared in a type declaration part in the outermost scope of a program or unit. Therefore, a class type can't be declared in a variable declaration part or within a procedure, function, or method block.
- Fields (instance variables) cannot be declared in an interface.
- Properties can only have procedures/functions for the getter and setter.

IUnknown and inheritance

The declaration of an interface may specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of *IUnknown*, the built-in ultimate ancestor of all interfaces. The group of methods identified by an interface consists of the methods declared by *IUnknown*, the methods declared by each ancestor interface, and the methods declared by the interface itself.

Inheritance in COM does not mean code reuse. Because implementations are not associated with interfaces, interface inheritance does not mean code inheritance. It means only that the contract associated with an interface is inherited in a pure-virtual base-class fashion and modified either by adding new methods or by further qualifying the allowed usage of methods. There is no selective inheritance in COM: If

one interface inherits from another, it includes all the methods that the other interface defines.

Inheritance is used sparingly in the predefined COM interfaces. All predefined interfaces inherit their definitions from the interface *IUnknown*, which contains three virtual methods: *QueryInterface*, *_AddRef*, and *_Release*. All COM objects must implement the *IUnknown* interface because it provides the means to move freely between the different interfaces that an object supports with *_QueryInterface*, and to manage its lifetime with *_AddRef* and *_Release*.

The *IUnknown* standard interface is declared by the *System* unit as follows:

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Interface identification

An interface can optionally include an *interface identifier* which consists of a string constant enclosed in square brackets immediately preceding the elements of the interface. The string constant given in an *interface identifier* must evaluate to a *globally unique identifier* (GUID) of the form

```
'{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}'
```

where each x is a hexadecimal digit (0 through 9 or A through F).

When an interface includes an *interface identifier*, it becomes possible to obtain references to implementations of the interface through *interface querying*. *Interface querying* is discussed further in "Interface querying" on page 13-13.

A *globally unique identifier*, also known as a GUID, is a 16-byte binary value that uniquely identifies an interface amongst all other interfaces.

The following types declared in the *System* unit are used to manipulate GUIDs.

```
type
    PGUID = ^TGUID;
    TGUID = record
      D1: Integer;
      D2: Word;
      D3: Word;
      D4: array[0..7] of Byte;
    end;
```

When declaring a typed constant of type TGUID, a string constant of the above form may be specified as the value of the GUID. For example,

```
const
  IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

When calling a procedure or function, an *interface identifier* may be specified in place of a parameter expression for a value or constant parameter of type TGUID. For example,

```
function Supports(Unknown: IUnknown; const IID: TGUID): Boolean;
```

can be called in the following fashion

```
if Supports(Allocator, IMalloc) then ...
```

Calling conventions

The default calling convention for methods of an interface is **register**.

Interfaces intended to be used between modules, in particular modules written in different programming languages, should declare all methods with **stdcall**.

The **safecall** calling convention is used to implement methods of dual interfaces. This is described further in "Dual interfaces" on page 13-16.

Member visibility

All members (methods and properties) of an interface have public visibility. The **private**, **protected**, **public**, **published**, and **automated** class visibility specifiers are not allowed in an interface.

Access specifiers

Properties declared in an interface must include a **read** access specifier, a **write** access specifier, or both. Since an interface cannot contain fields, the access specifiers are only allowed to reference methods of the interface. A **default** directive may be used to mark an array property as the default property of the interface.

Forward declarations

An interface declaration consisting only of the reserved word **interface** is a forward declaration. A forward declaration must be resolved by a normal declaration of the interface within the same type declaration part. Forward declarations allow mutually dependent interfaces to be declared. For example,

```
type
  IControl = interface;
  IWindow = interface
    ['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
    function GetControl(Index: Integer): IControl;
    ...
  end;
  IControl = interface
    ['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
    function GetWindow: IWindow;
```

```
  ...
end;
```

Mutually derived interfaces are not allowed. For example, `IWindow(IControl)` followed by `IControl(IWindow)` is not legal.

Interface implementations

Once an interface has been declared it must be implemented in a class before it can be used. Any Object Pascal class can implement interfaces.

Class declaration

The interfaces implemented by a class are specified by following the ancestor class identifier with a comma-separated list of interface identifiers. For example,

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ...
  end;
```

For example the *TInterfacedObject* class, which implements the *IUnknown* interface, is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IUnknown)
  private
    FRefCount: Integer;
  protected
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    property RefCount: Integer read FRefCount;
  end;
```

TInterfacedObject descends directly from *TObject*, a class that does not implement interfaces.

Classes that implement interfaces can also be used as base classes. For example, the following declaration:

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ...
  end;
```

specifies that the *TMemoryManager* class is a direct descendent of *TInterfacedObject* and implements the *IMalloc* and *IErrorInfo* interfaces.

Interface method implementation

When a class specifies one or more interfaces in ancestor list, that class must provide the implementation for the methods of each interface. To implement the interface, the interface methods must be either implicitly or explicitly mapped onto compatible methods in the class.

Mapping interface methods

An interface is implemented by mapping the methods of the interface onto compatible methods in the class. A class can successfully implement an interface only if each and every method of the interface can be mapped onto a compatible method in the class.

For an interface method to be considered compatible with a class method, the methods must adhere to the following rules:

- methods must have the same number of parameters
- parameters in corresponding positions must be of identical types
- the result types (if any) must be identical
- the calling conventions must be the same

By default, each method in an interface is mapped onto a compatible method of the same name in the implementing class. For example, given the following declarations of the *IMalloc* interface:

```
type
  IMalloc = interface(IUnknown)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

TMemoryManager must implement or inherit the following methods for it to successfully implement *IMalloc*.

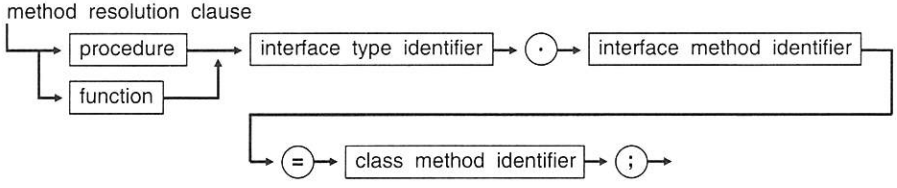
```
function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
function _AddRef: Integer; stdcall;
function _Release: Integer; stdcall;
function Alloc(Size: Integer): Pointer; stdcall;
function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
procedure Free(P: Pointer); stdcall;
function GetSize(P: Pointer): Integer; stdcall;
function DidAlloc(P: Pointer): Integer; stdcall;
procedure HeapMinimize; stdcall;
```

Since every interface inherits from the IUnknown standard interface, an object that implements interfaces is always required to implement the *QueryInterface*, *_AddRef*,

and *_Release* methods defined by *IUnknown*. The *TInterfacedObject* class defined in the *System* unit provides an implementation of these methods and is thus a convenient base class for classes that implement interfaces.

Method resolution clauses

The default name-based interface method mapping can be overridden by including interface method resolution clauses in the class declaration. Such clauses allow interface methods to be mapped onto compatible class methods with differing names.



For example, the declaration

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ...
  end;
```

specifies that the *Alloc* and *Free* methods of the *IMalloc* interface should map onto the *Allocate* and *Deallocate* methods of *TMemoryManager*.

A class method identifier specified in an interface method resolution clause must name a method in the class or an ancestor of the class. In effect this means that an interface method resolution clause is allowed to make a forward reference to a method identifier that is declared later on within the same class declaration. For example, in the following declaration:

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ...
    function Allocate;
    procedure Deallocate;
  end;
```

the *TMemoryManager* class methods, *Allocate* and *Deallocate*, are declared after the interface method resolution clauses.

When two or more interfaces have a method with the same name, use the method resolution clause to resolve name conflicts. Declare and implement a new method under a different name, and then map the interface method to this new method. For example,

```
type
  IWindow = interface
```

```
  procedure Draw;
  ...
end;

ICanvas = interface
  procedure Draw;
  ...
end;

TWindow = class(TInterfacedObject, IWindow, ICanvas)
  procedure IWindow.Draw = Drawing;
  ...
  procedure Drawing;
  procedure Draw;
 end;
```

The *IWindow* interface *Draw* method is mapped to a method of a different name in *TWindow*. The *Drawing* method of *TWindow* implements the *Draw* method of *IWindow*, whereas the *Draw* method of *TWindow* implements the *Draw* method of the *ICanvas* interface.

Method resolution clause restrictions

There are several restrictions of method resolution clauses that apply when mapping methods in inherited classes. These restrictions deal with the affect of the mapping and type of methods in the base class, and the reimplementation of the interface in the inherited class. The restictions are:

1 An interface method cannot be mapped onto an inherited private method of an ancestor class declared in another module.

  When mapping interface methods onto class methods, the compiler excludes any methods that are not visible in the class being implemented. This in effect means that an interface method cannot be mapped onto an inherited private method of an ancestor class declared in another module.

2 interface method resolution clauses cannot alter the mapping of methods of interfaces that were introduced by ancestors of the class.

  The mapping of an interface method onto a method of a class is established once and for all in the class that introduces the interface, and interface method resolution clauses can only be applied to interfaces that are introduced by the class in which the interface method resolution clauses are listed.

3 Subclasses are allowed to change the implementation of an interface method only if the interface method was mapped onto a virtual or dynamic method

  When a virtual or dynamic method is overridden, then all interface methods that are mapped onto the virtual or dynamic method are affected by the override. Only by mapping an interface method onto a virtual or dynamic method can a class allow subclasses to change the implementation of the interface method. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
```

```
    procedure Draw;
    ...
  end;
  TWindow = class(TInterfacedObject, IWindow)
    procedure Draw; virtual;
    ...
  end;
  TFrameWindow = class(TWindow)
    procedure Draw; override;
    ...
  end;
```

Here, a call to the *Draw* method of an *IWindow* interface obtained from an instance of *TWindow* will invoke *TWindow.Draw*, and a call to the *Draw* method of an *IWindow* interface obtained from an instance of *TFrameWindow* will invoke *TFrameWindow.Draw*.

4 A class is allowed to reimplement an interface that was inherited from a base class

When a class implements a particular interface, that implementation effectively hides any inherited implementation of the same interface. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ...
  end;
  TWindow = class(TInterfacedObject, IWindow)
    procedure Draw;
    ...
  end;
  TFrameWindow = class(TWindow, IWindow)
    procedure Draw;
    ...
  end;
```

Here, *TFrameWindow*'s implementation of the *IWindow* interface hides the implementation that was inherited from *TWindow*. Also, since *TFrameWindow*'s *Draw* method hides the *Draw* method that was inherited from *TWindow*, each *IWindow* interface will have its own implementation of *Draw*.

5 Inherited interfaces are not affected by reimplementations

The reimplementation of an interface is completely separate from the inherited implementation that it hides, and the interface method resolution clauses (if any) that were used to resolve the inherited interface have no effect on the reimplemented interface.

When a class implements an interface, it *does not* implicitly implement any ancestor interfaces of that interface. A class *only* implements the interfaces that are specifically named in the class declaration and any ancestor class declarations.

```
type
  IControl = interface
    ...
```

```
end;
IWindow = interface(IControl)
  ...
end;
TWindow = class(TInterfacedObject, IWindow)
  ...
end;
```

Here, since *IWindow* is a descendant of *IControl*, an *IWindow* interface instance can also function as an *IControl* instance. However, given that *TWindow* does not specifically list *IControl* as an implemented interface, a request for a *TWindow*'s *IControl* interface would fail.

Since every interface inherits from the *IUnknown* standard interface, an object that implements interfaces is always required to implement the *QueryInterface*, *_AddRef*, and *_Release* methods defined by *IUnknown*. The *TInterfacedObject* class defined in the *System* unit provides an implementation of these methods and is thus a convenient base class for classes that implement interfaces. *TInterfacedObject* is described further in a following section.

Interface references

A variable can be declared that is a reference to an interface. An interface reference is a reference to an implementation of an interface. Through an interface reference it is possible to invoke methods of a given interface without any knowledge of the actual implementation of the interface.

Reference counting

Interface references are *reference counted*. Through reference counting it is possible for the implementation of an interface to know how many references exist to the interface. Further, reference counting makes it possible for the implementation of an interface to dispose of the underlying object(s) once the reference count reaches zero.

Interface reference counting is entirely automatic in Object Pascal, and is implemented by calling the *_AddRef* and *_Release* methods that every interface inherits from *IUnknown*. When an interface variable is assigned a new interface reference, the reference count of the previous interface reference (if any) is automatically decremented by calling *_Release*, and the reference count of the new interface reference (if any) is automatically incremented by calling *_AddRef*. Likewise, when an interface variable goes out of scope, the reference count of the referenced interface is automatically decremented by calling *_Release*.

Type compatibility

Two interface types are *compatible* if they are identical or if one is derived from the other. Furthermore, the value **nil** is compatible with any interface type. The relational operators = and <> can be used to compare compatible interface types.

An interface type T2 is *assignment compatible* with an interface type T1 (that is, T1 := T2 is allowed) if T2 is identical to or derived from T1. In other words, a reference of a particular interface type can also function as a reference of a base type of that interface type.

A class type is assignment compatible with any interface type implemented by the class type. In other words, if a class type T implements (or inherits implementations of) interface types I1 and I2, then T is assignment compatible with I1 and I2.

Any interface type is assignment compatible with the *Variant* type. If the interface is of type *IDispatch* or a descendant, the resulting *Variant* value has a type code of *varDispatch*. Otherwise, the resulting *Variant* value has a type code of *varUnknown*.

The *Variant* type is assignment compatible with the *IUnknown* and *IDispatch* types. The type code of a Variant must be *varEmpty*, *varUnknown*, or *varDispatch* for it to be successfully converted to an *IUnknown*, and *varEmpty* or *varDispatch* for it to be successfully converted to an *IDispatch*. Otherwise, an exception is raised.

Interface conversions

Interface types follow the same rules as class types in variable and value typecasts.

A value of a class type can be explicitly cast to an interface type using a typecast of the form *IntfType(X)*, where *X* is an expression of a class type and *IntfType* is the type identifier of an interface implemented by the class type.

A value of an interface type can be explicitly cast to type *Variant* using a typecast of the form *Variant(X)*, where *X* is an expression of an interface type. If *X* is of type *IDispatch* or a descendant, the resulting variant type code is *varDispatch*. Otherwise the resulting type code is *varUnknown*.

A *Variant* value can be explicitly cast to type *IUnknown* and type *IDispatch* using typecasts of the form *IUnknown(X)* and *IDispatch(X)*, where *X* is an expression of type *Variant*. The type code of a *Variant* must be *varEmpty*, *varUnknown*, or *varDispatch* for it to be successfully converted to an *IUnknown*, and *varEmpty* or *varDispatch* for it to be successfully converted to an *IDispatch*. Otherwise, an exception is raised.

Using assignment to obtain an interface reference

As stated in the type compatibility rules above, an object is assignment compatible with any interfaces implemented by the class of the object. This means that references to interfaces on an object can be obtained through simple assignments. For example,

```
type
  IWindow = interface
    ...
  end;
  IDragDrop = interface
    ...
  end;
  TEditWindow = class(TObject, IWindow, IDragDrop)
    ...
```

```
  end;

var
  EditWindow: TEditWindow;
  Window: IWindow;
  DragDrop: IDragDrop;
```

Given the above declarations, simple assignments can be used to obtain the *IWindow* and *IDragDrop* interfaces of a *TEditWindow* object.

```
Window := EditWindow;
DragDrop := EditWindow;
```

Attempting to convert an object to an interface type that isn't implemented by the object produces a compile-time error. Note that the conversion of an object to an interface type is based on the *declared* type of the object, not the *actual* type. For example,

```
var
  Instance: TObject;
  Window: IWindow;
begin
  Instance := TEditWindow.Create(...);
  Window := Instance;
  ...
end;
```

The above assignment of *Instance* to *Window* produces a compile-time error because the declared type of *Instance* is *TObject* (which doesn't implement *IWindow*) even though the actual type of *Instance* is *TEditWindow*. The **as** operator can be used to obtain an interface from an object based on the actual type of the object. This is described in a following section.

In cases where an application interacts with an object solely through interfaces, the lifetime of the object is effectively controlled by the automatic reference counting of its interfaces. This means that there is no need to manually destroy the object—it will be automatically destroyed when the last interface reference goes out of scope. For example,

```
procedure CreateEditWindow;
var
  Window: IWindow;
begin
  Window := TEditWindow.Create(...);
  ...
end;
```

In the above example, upon returning, the *CreateEditWindow* procedure will automatically call *Window._Release* which, assuming that no other interfaces have been obtained from the object, will destroy the *TEditWindow* object.

Interface querying

Interface references can be produced from object references and from other interface references through *interface querying* using the **as** operator. An interface querying

operation takes the form *Reference* **as** *Interface*, where *Reference* is an expression of a class type that implements the *IUnknown* interface or an expression of any interface type, and *Interface* is an interface type identifier. The result of such an operation is an interface reference of the given interface type. If the object or interface being queried does not implement the given interface, an exception is raised.

The **as** operator can be used to query an object for an interface only if the object implements the *IUnknown* interface. An operation of the form *Reference* **as** *Interface*, where *Reference* is an object, is effectively transformed into *IUnknown*(*Reference*) **as** *Interface*, which will only succeed if the object implements *IUnknown*.

The operation *Reference* **as** *Interface* is evaluated as follows

- If *Reference* is **nil**, the resulting value is **nil**.
- If *Reference* is not **nil**, the *QueryInterface* method of *Reference* is called with the *GUID* of the *Interface* type. If *QueryInterface* returns a non-zero value, indicating failure, an exception is raised. Otherwise, the resulting value is the new interface reference returned by *QueryInterface*.

Interactions with other language features

It is possible to declare initialized variables of interface types, but the only allowed initial value for such variables is **nil**.

The *Assigned* standard function may be passed an expression of an interface type. The result is *False* if the interface is **nil**, or *True* otherwise.

A **with** statement may specify an expression of an interface type.

An expression of an interface type may be used as an element of a type variant open array constructor. The corresponding value type code of such an element is *vtInterface*.

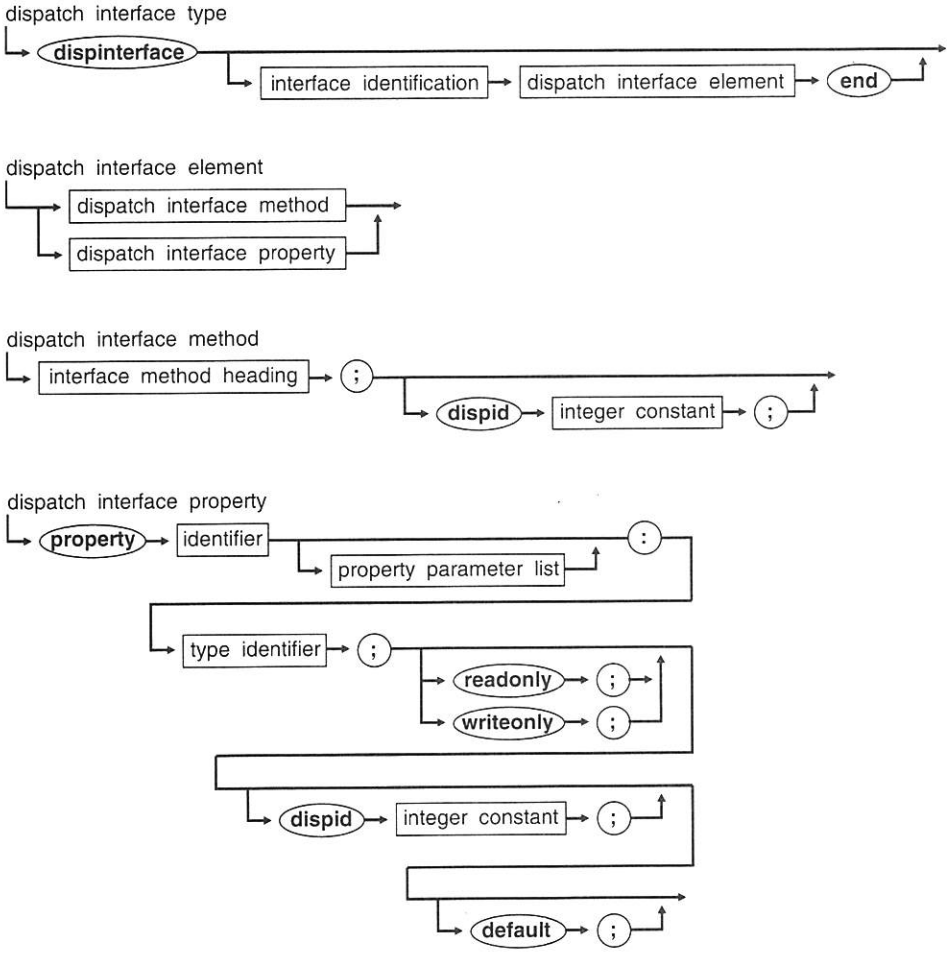
An OLE Automation method call parameter can be of an interface type.

OLE automation Interface types

Objects that implement the *IDispatch* interface are OLE automation objects. OLE automation introduces the concept of dispatch interfaces and dual interfaces.

Dispatch interfaces

Dispatch interface declarations are used to describe the methods and properties an OLE Automation object implements through its *IDispatch* interface. Whereas regular interfaces use direct v-table dispatching, calls to methods of a dispatch interface are routed through the *Invoke* method of the underlying *IDispatch* implementation.



Dispatch interfaces are characterized by the following restrictions:

- It is not possible for a class to list a dispatch interface as one of its implemented interfaces. Since methods of dispatch interfaces are really just prototypes for calls to the *Invoke* method of the underlying *IDispatch* implementation, dispatch interfaces are never actually implemented.
- The declaration of a dispatch interface cannot specify an ancestor interface.
- Method declarations in a dispatch interface do not allow any interface method directives such as **stdcall** to be specified.
- All function result and parameter types must be OLE Automation compatible.

Dual interfaces

A *dual interface* is an interface that supports both compile-time v-table binding and runtime dynamic binding through OLE Automation. A dual interface must descend (directly or indirectly) from the *IDispatch* interface, which is declared in the *System* unit as follows:

```
type
  IDispatch = interface(IUnknown)
    function GetTypeInfoCount(out Count: Integer): Integer; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): Integer; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): Integer; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word;
      Params, VarResult, ExcepInfo, ArgErr: Pointer): Integer; stdcall;
  end;
```

All methods of a dual interface (except from those inherited from *IUnknown* and *IDispatch*) must use the **safecall** calling convention, and all function result and parameter types must be OLE Automation compatible. Methods declared as **safecall** automatically implement OLE conventions for errors and exception handling.

Chapter

14

Dynamic-link libraries

Dynamic-link libraries (DLLs) permit several Windows applications to share code and resources. With Object Pascal, you can use DLLs as well as write your own DLLs to be used by other applications.

What is a DLL?

A DLL is an executable module containing code or resources for use by other applications or DLLs. Conceptually, a DLL is similar to a unit—both have the ability to provide services in the form of procedures and functions to a program. There are, however, many differences between DLLs and units. In particular, units are *statically linked*, whereas DLLs are *dynamically linked*.

When a program uses a procedure or function from a unit, a copy of that procedure or function's code is statically linked into the program's executable file. If two programs are running simultaneously and they use the same procedure or function from a unit, there will be two copies of that routine present in the system. It would be more efficient if the two programs could share a single copy of the routine. DLLs provide that ability.

In contrast to a unit, the code in a DLL isn't linked into a program that uses the DLL. Instead, a DLL's code and resources are in a separate file containing binary executable code with a .DLL extension. This file must be present when the client program runs. The procedure and function calls in the program are dynamically linked to their entry points in the DLLs used by the application.

Another difference between units and DLLs is that units can export types, constants, data, and objects whereas DLLs can export procedures and functions only.

Note A DLL doesn't have to be written in Object Pascal for an Object Pascal application to be able to use it. Also, programs written in other languages can use DLLs written in Object Pascal. DLLs are therefore ideal for multi-language programming projects.

Using DLLs

For a module to use a procedure or function in a DLL statically, the module must import the procedure or function using an external declaration. For example, the following **external** declaration imports a function called *MessageBox* from the DLL called "user32.dll" (part of the Windows API):

```
function MessageBox(HWnd: Integer; Text, Caption: PChar;
  Flags: Integer): Integer; stdcall;
  external 'user32.dll' name 'MessageBoxA';
```

In imported procedures and functions, the **external** directive takes the place of the declaration and statement parts that would otherwise be present, but otherwise imported procedures and functions behave no differently than normal procedures and functions. See "External declarations" on page 9-4.

Object Pascal imports procedures and functions in three ways:

- By name
- By new name
- By ordinal

The format of **external** directives for each of the three methods is demonstrated in the following examples.

When no **index** or **name** clause is specified, the procedure or function is imported by name. The name used is the same as the procedure or function's identifier, *with the same spelling and case*. In this example, the *ImportByName* procedure is imported from 'testlib.dll' using the name 'ImportByName'.

```
procedure ImportByName; external 'testlib.dll';
```

When a **name** clause is specified, the procedure or function is imported by a different name than its identifier. Here the *ImportByNewName* procedure is imported from 'testlib.dll' using the name 'RealName':

```
procedure ImportByNewName; external 'testlib.dll' name 'RealName';
```

Finally, when an **index** clause is present, the procedure or function is imported by ordinal. Importing by ordinal reduces the module's load time because the name doesn't have to be looked up in the DLL's name table. In the example, the *ImportByOrdinal* procedure is imported as the fifth entry in 'testlib.dll':

```
procedure ImportByOrdinal; external 'testlib.dll' index 5;
```

The DLL name specified after the **external** keyword and the new name specified in a **name** clause don't have to be string literals. Any constant-string expression is allowed. Likewise, the ordinal number specified in an **index** clause can be any constant-integer expression.

```
const
  TestLib = 'testlib.dll';
  Ordinal = 5;

procedure ImportByName;    external TestLib;
procedure ImportByNewName; external TestLib name 'RealName';
procedure ImportByOrdinal; external TestLib index Ordinal;
```

Although a DLL can have variables, it's not possible to import them in other modules. Any access to a DLL's variables must take place through a procedural interface.

Import units

Declarations of imported procedures and functions can be placed directly in the program that imports them. Usually, though, they are grouped together in an *import unit* that contains declarations for all procedures and functions in a DLL, along with any constants and types required to interface with the DLL. The *Windows* unit supplied with Delphi is an example of such an import unit. Import units aren't a requirement of the DLL interface, but they do simplify maintenance of projects that use multiple DLLs.

As an example, consider a DLL called DATETIME.DLL that has four routines to get and set the date and time, using a record type that contains the day, month, and year, and another record type that contains the second, minute, and hour. Instead of specifying the associated procedure, function, and type declarations in every program that uses the DLL, you can construct an import unit to go along with the DLL. This code creates a .DCU file, but it doesn't contribute code or data to the programs that use it:

```
unit DateTime;

interface

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

type
  TDateRec = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

procedure SetTime(var Time: TTimeRec);
procedure GetTime(var Time: TTimeRec);
procedure SetDate(var Date: TDateRec);
procedure GetDate(var Date: TDateRec);

implementation

procedure SetTime; external 'DATETIME' index 1;
procedure GetTime; external 'DATETIME' index 2;
procedure SetDate; external 'DATETIME' index 3;
procedure GetDate; external 'DATETIME' index 4;

end.
```

Any program that uses DATETIME.DLL can now simply specify *DateTime* in its **uses** clause. Here is a Windows program example:

```
program ShowTime;
{ $APPTYPE CONSOLE }

uses DateTime;

var
  Time: TTimeRec;

begin
  GetTime(Time);
  with Time do
    WriteLn('The time is ', Hour, ':', Minute, ':', Second);
end.
```

Another advantage of an import unit such as *DateTime* is that when the associated DATETIME.DLL is modified, only one unit, the *DateTime* import unit, needs updating to reflect the changes.

When you compile a program that uses a DLL, the compiler doesn't look for the DLL so it need not be present. The DLL must be present when you run the program, however.

Note If you write your own DLLs, they aren't automatically compiled when you compile a program that uses the DLL. Instead, DLLs must be compiled separately.

Static and dynamic imports

The **external** directive provides the ability to statically import procedures and functions from a DLL. A statically imported procedure or function always refers to the same entry point in the same DLL. Windows also supports dynamic imports, whereby the DLL name and the name or ordinal number of the imported procedure or function is specified at run time. The *ShowTime* program shown here uses dynamic importing to call the *GetTime* procedure in DATETIME.DLL. Note the use of a procedural-type variable to represent the address of the *GetTime* procedure.

```
program ShowTime;

uses Windows;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;

begin
  Handle := LoadLibrary('DATETIME.DLL');
```

```
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

Writing DLLs

The structure of an Object Pascal DLL is identical to that of a program, except a DLL starts with a **library** header instead of a **program** header. The **library** header tells Object Pascal to produce an executable file with the extension .DLL instead of .EXE, and also ensures that the executable file is marked as being a DLL.

The example here implements a very simple DLL with two exported functions, *Min* and *Max*, that calculate the smaller and larger of two integer values.

```
library MinMax;

function Min(X, Y: Integer): Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min index 1,
  Max index 2;

begin
end.
```

Note the use of the **stdcall** calling convention for the exported functions. Although not a requirement, use of the **stdcall** calling convention makes it possible for applications written in other languages to use the DLL. Other languages may not support the default **register** calling convention used by Delphi. Also note the use of

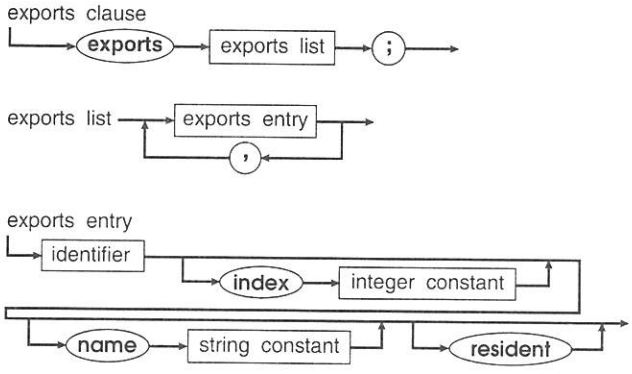
an **exports** clause to export the two routines, supplying an optional ordinal number for each of them.

Although the preceding example doesn't demonstrate it, libraries can and often do consist of several units. In such cases, the library source file itself is frequently reduced to a **uses** clause, an **exports** clause, and the library's initialization code. For example,

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors index 1,
  DoneEditors index 2,
  InsertText index 3,
  DeleteSelection index 4,
  FormatSelection index 5,
  PrintSelection index 6,
   :
  SetErrorHandler index 53;

begin
  InitLibrary;
end.
```

The exports clause

A procedure or function is exported by a DLL when it's listed in the library's **exports** clause.



An **exports** clause can appear anywhere and any number of times in a program or library's declaration part. Each entry in an **exports** clause specifies the identifier of a procedure or function to be exported. That procedure or function must be declared before the **exports** clause appears. You can precede the identifier in the **exports** clause with a unit identifier and a period; this is known as a fully qualified identifier.

An exports entry can include an **index** clause, which consists of the word **index** followed by a cardinal constant between 1 and 2,147,483,647. However, since index is a literal value into a table that is created, it is recommended that you use index values

that are low. When an **index** clause is specified, the procedure or function to be exported uses the specified ordinal number. If no **index** clause is present in an exports entry, an ordinal number is automatically assigned. The quickest way to look up a DLL entry is by index.

An entry can also have a **name** clause, which consists of the word **name** followed by a string constant. When there is a **name** clause, the procedure or function to be exported uses the name specified by the string constant. If no **name** clause is present in an exports entry, the procedure or function is exported by its identifier with the same spelling and case.

A program can contain an **exports** clause, but it seldom does because Windows doesn't allow application modules to export functions for use by other applications.

Library initialization code

The statement part of a library constitutes the library's *initialization code*. The initialization code is executed once, when the library is initially loaded.

A library's initialization code typically performs tasks like registering window classes for window procedures contained in the library, and setting initial values for the library's global variables. In addition, the initialization code of a library can install an exit procedure using the *ExitProc* variable as described in Chapter 19, "Control issues". The exit procedure will be executed when the operating system unloads the library.

The initialization code of a library can signal an error condition by setting the *ExitCode* variable to a non zero value. *ExitCode* is declared in the *System* unit and defaults to zero, indicating successful initialization. If the initialization code sets *ExitCode* to a non zero value, the DLL is unloaded from memory and the calling application is notified of the failure to load the DLL.

Note If an unhandled exception escapes from the initialization code while executing a library's initialization code, the calling application will be notified of a failure to load the DLL.

Here is an example of a library with initialization code and an exit procedure:

```
library Test;
var
  SaveExit: Pointer;
procedure LibExit;
begin
  :
  { Library exit code }
  :
  ExitProc := SaveExit;{ Restore exit procedure chain }
end;
begin
  :
  { Library initialization code }
  :
  SaveExit := ExitProc;{ Save exit procedure chain }
```

```
  ExitProc := @LibExit;{ Install LibExit exit procedure }
end.
```

When a DLL is unloaded, Delphi executes the library's exit procedures by continuing to call the address stored in the *ExitProc* variable until *ExitProc* becomes **nil**. Because this works the same way exit procedures are handled in Object Pascal programs, you can use the same exit procedure logic in both programs and libraries.

Note The **initialization** parts of all units used by an application or library are always executed *before* the application or library's statement part. Likewise, unit **finalization** parts are executed *after* an exit procedure declared by an application or library. Unit **finalization** parts in fact use the *ExitProc* variable to install themselves.

Library programming notes

The following sections note important points you should keep in mind while working with DLLs.

Global variables in a DLL

Global variables declared in a DLL are private to that DLL. A DLL cannot access variables declared by modules that call the DLL, and it is not possible for a DLL to export its variables for use by other modules. Such access must take place through a procedural interface.

Even though a DLL can be used by multiple applications at the same time, to the DLL it appears that there is only one client, and each instance of the DLL will have its own set of global variables. For multiple DLLs (or multiple instances of one DLL) to share memory, the DLLs must use memory mapped files. Refer to the Windows API documentation for further details on this topic.

DLLs and the System unit

The *IsLibrary* Boolean variable can be used to determine whether code is executing in the context of an application or a library. *IsLibrary* is always *False* in an application, and always *True* in a library. During a DLL's lifetime, the *HInstance* variable contains the instance handle of the DLL. The *CmdLine* variable is always **nil** in a DLL.

Exceptions and runtime errors in DLLs

If an exception is raised but not handled in a DLL, the exception is propagated out of the DLL. If the calling application or DLL is itself written in Delphi, it is possible to handle the exception through a normal **try...except** statement.

If the calling application or DLL is written in another programming language, the exception can be handled as an operating system exception with an exception code of $0EEDFACE. The first entry in the *ExceptionInformation* array of the operating system

exception record contains the exception address, and the second entry contains a reference to the Delphi exception object.

If a DLL does not use the *SysUtils* unit, Delphi's exception support is disabled. In that case, if a runtime error occurs in a DLL, the application that called the DLL terminates. Because a DLL has no way of knowing whether it was called from an Object Pascal application or an application written in another programming language, it is not possible for the DLL to invoke the application's exit procedures before the application is terminated. The application is simply aborted and removed from memory. For this reason, make sure that there are sufficient checks in any DLL code so such errors do not occur.

The Shared Memory Manager

If a DLL exports any procedures or functions that pass long strings as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the *ShareMem* unit. The same is true if one module (application or DLL) allocates memory with *New* or *GetMem* which is deallocated by a call to *Dispose* or *FreeMem* in another module.

ShareMem is the interface unit for the DELPHIMM.DLL shared memory manager, which must be deployed along with applications and/or libraries that use *ShareMem*. When an application or DLL uses *ShareMem*, the application or DLL's memory manager is replaced by the memory manager in DELPHIMM.DLL, making it possible to share dynamically allocated memory between multiple modules.

When used by an application or library, the *ShareMem* unit should be the first unit listed in an application or library's **uses** clause.

The DLLProc variable

The *DLLProc* variable defined in the *System* unit allows a DLL to monitor all calls that the operating system makes to the DLL's entry point. This feature is normally only of interest to DLLs that support multi-threading.

To monitor operating system calls to a DLL's entry point, assign the address of procedure with the following parameter list to the *DLLProc* variable.

```
procedure DLLHandler(Reason: Integer);
```

When the DLL procedure is invoked, the *Reason* parameter will contain one of the following values (defined in the *Windows* unit):

DLL_PROCESS_DETACH	Indicates that the DLL is detaching from the address space of the calling process as a result of either a clean process exit or of a call to *FreeLibrary*.
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread.
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly.

Chapter

15

Packages

A package is a specially compiled dynamic-link library used by Delphi applications, the Delphi IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in Delphi's IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their **requires** clauses.

To distinguish them from other DLLs, package libraries are stored in files that end with the .DPL (Delphi package library) extension.

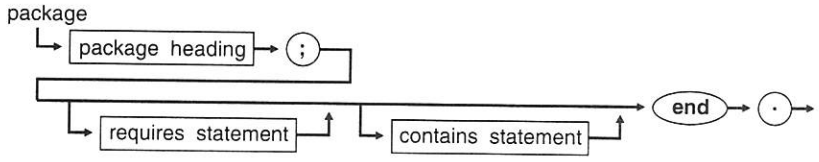
Note When an application utilizes packages, the name of each packaged unit still must appear in the **uses** clause of any source file that references it.

Package Source Files

Package source files should be saved with the .DPK (Delphi package) extension to avoid confusion with other files containing Object Pascal source code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains

- A *name* for the package.
- A list of other packages *required* by the new package. These are packages to which the new package is linked.
- A list of unit files *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled DPL.

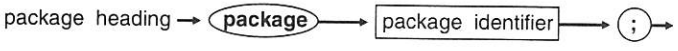
The following diagram summarizes the syntax for a package source file.



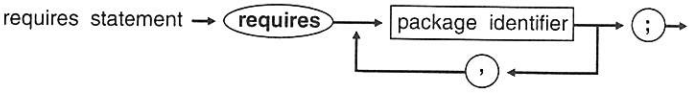
For example, the following code declares the VCLDB30 package.

```
package VCLDB30;
   requires VCL30;
   contains Db, Dbcgrids, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables;
end.
```

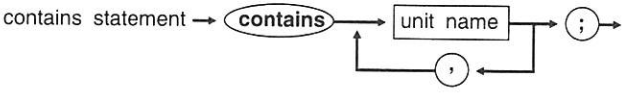
The package heading specifies the name for the package.



The **requires** clause lists other, external packages used by the current package. If a package does not reference other packages, then it does not need a **requires** clause.



The **contains** clause identifies the unit files to be compiled and bound into the package.



Naming Packages

A compiled package involves several generated files. For example, the source file for the package called VCL30 is VCL30.DPK, from which the compiler generates an executable and a binary image called VCL30.DPL and VCL30.DCP, respectively. "VCL30" is used to refer to the package in the **requires** clauses of other packages, or when using the package in an application. Package names must be unique within a project.

The requires clause

The **requires** clause lists other, external packages that are used by the current package. It functions like the **uses** clause in a unit file. An external package listed in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's **requires** clause. If the other

packages are omitted from the **requires** clause, the compiler loads the referenced units from their DCU files.

Avoiding circular package references

Packages cannot contain circular references in their **requires** clauses. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Duplicate package references

The compiler ignores duplicate references in a package's **requires** clause. For programming clarity and readability, however, duplicate references should be removed.

The contains clause

The **contains** clause identifies the unit files to be bound into the package. Do not include file-name extensions in the **contains** clause.

Avoiding redundant source code uses

A package cannot be listed in the **contains** clause of another package or the **uses** clause of a unit.

All units included directly in a package's **contains** clause, or indirectly in the **uses** clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in **requires** clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

Compiling packages

Packages are ordinarily compiled from the Delphi IDE using .DPK files generated by the Package editor. You can also compile .DPK files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled if necessary.

Generated files

The following table lists the files produced by the successful compilation of a package.

Table 15.1 Compiled package files

File extension	Contents
DCP	A binary image containing a package header and the concatenation of all DCU files in the package. A single DCP file is created for each package. The base name for the DCP is the base name of the DPK source file.
DCU	A binary image for a unit file contained in a package. One DCU is created, when necessary, for each unit file.
DPL	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the DPL is the base name of the DPK source file.

Several compiler directives and command-line switches are available to support package compilation.

Package-specific compiler directives

The following table lists package-specific compiler directives that can be inserted into source code. See the *User's Guide* for details.

Table 15.2 Package-specific compiler directives

Directive	Purpose
{$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .DPK files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{$G–} or {IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit "weakly", as explained in the *User's Guide*.
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the Delphi IDE. (Put in .DPK file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .DPK file.)

Including **{$DENYPACKAGEUNIT ON}** in source code prevents the unit file from being packaged. Including **{$G–}** or **{IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the Delphi IDE.

Other compiler directives may be included, if appropriate, in package source code.

Package-specific command-line compiler switches

The following package-specific switches are available for the command-line compiler. See the *User's Guide* for details.

Table 15.3 Package-specific command-line compiler switches

Switch	Purpose
-$G–	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
-LE *path*	Specifies the directory where the package DPL file will be placed.
-LN *path*	Specifies the directory where the package DCP file will be placed.
-LU*packageName* [;*packageName2;...*]	Specifies additional runtime packages to use in an application. Used when compiling a project.
-Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the **-$G–** switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

Chapter

16

Input and output

This chapter describes Delphi's standard (or built-in) input and output procedures and functions. You'll find them in the *System* unit.

Table 16.1 Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what *Read* does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.

Table 16.1 Input and output procedures and functions (continued)

Procedure or function	Description
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as *Write*, and then writes an end-of-line marker to the text file.

File input and output

An Object Pascal file variable is any variable whose type is a file type. There are three classes of Object Pascal files: *typed*, *text*, and *untyped*. The syntax for writing file types is given in "File types" on page 4-15.

Before a file variable can be used, it must be associated with an external file through a call to the *AssignFile* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be "opened" to prepare it for input or output. An existing file can be opened via the *Reset* procedure, and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. Each component has a component number. The first component of a file is considered to be component zero.

Files are normally accessed *sequentially*; that is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write*, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly, however, using the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure *CloseFile*. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be

turned on and off using the {**$I+**} and {**$I–**} compiler directives. When I/O checking is off—that is, when a procedure or function call is compiled in the {**$I–**} state—an I/O error doesn't cause an exception to be raised. To check the result of an I/O operation, you must call the standard function *IOResult* instead.

You must call the *IOResult* function to clear whatever error may have occurred, even if you aren't interested in the error. If you don't and {**$I+**} is the current state, the next I/O function call fails with the lingering *IOResult* error.

Text files

This section summarizes I/O using file variables of the standard type *Text*.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character). In Object Pascal, the type *Text* is distinct from the type **file of** *Char*.

For text files, there are special forms of *Read* and *Write* that let you read and write values that are not of type *Char*. Such values are automatically translated to and from their character representation. For example, *Read(F, I)*, where *I* is a type *Integer* variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in *I*.

Object Pascal defines two standard text-file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input file (typically the keyboard). The standard file variable *Output* is a write-only file associated with the operating system's standard output file (typically the display). Before an application begins executing, *Input* and *Output* are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Note Text-oriented I/O is only available in console applications, that is applications compiled with the "Generate console application" option checked on the Linker page of the Project Options dialog box or with the **-cc** command-line compiler option. In a GUI (non-console) application, any attempt to read or write using *Input* or *Output* will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, *Read(X)* corresponds to *Read(Input, X)* and *Write(X)* corresponds to *Write(Output, X)*.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using *AssignFile*, and opened using *Reset*, *Rewrite*, or *Append*. An exception is raised if you pass a file that was opened with *Reset* to an output-oriented procedure or function. An exception is

also raised if you pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

Untyped files

Untyped files are low-level I/O channels primarily used for direct access to any disk file regardless of type and structuring. An untyped file is declared with the word **file** and nothing more. For example,

```
var
  DataFile: file;
```

For untyped files, the *Reset* and *Rewrite* procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file (no partial records are possible when the record size is 1).

Except for *Read* and *Write*, all typed-file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

Text-file device drivers

Object Pascal lets you define your own text-file device drivers for your Windows programs. A text-file device driver is a set of four functions that completely implement an interface between Object Pascal's file system and some device.

The four functions that define each device driver are *Open*, *InOut*, *Flush*, and *Close*. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where *TTextRec* is the text-file record-type defined in "File types" on page 18-7. The return value of a device-interface function becomes the value returned by *IOResult*. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized *Assign* procedure. The *Assign* procedure must assign the addresses of the four device-interface functions to the four function pointers in the text-file variable. In addition, it should store the *fmClosed* "magic" constant in the *Mode* field, store the size of the text-file buffer in *BufSize*, store a pointer to the text-file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device-interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
```

```
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device-interface functions can use the *UserData* field in the file record to store private information. This field isn't modified by the Delphi file system at any time.

The Open function

The *Open* function is called by the *Reset*, *Rewrite*, and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput*, *fmOutput*, or *fmInOut* to indicate whether the *Open* function was called from *Reset*, *Rewrite*, or *Append*.

The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutput* before *Open* returns.

Open is always called before any of the other device-interface functions. For that reason, *AssignDev* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode*, *Open* can then install pointers to either input- or output-oriented functions. This saves the *InOut*, *Flush* functions and the *CloseFile* procedure from determining the current mode.

The InOut function

The *InOut* function is called by the *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln*, and *CloseFile* standard procedures and functions whenever input or output from the device is required.

When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into *BufPtr^*, and returns the number of characters read in *BufEnd*. In addition, it stores zero in *BufPos*. If the *InOut* function returns zero in *BufEnd* as a result of an input request, *Eof* becomes *True* for the file.

When *Mode* is *fmOutput*, the *InOut* function writes *BufPos* characters from *BufPtr^*, and returns zero in *BufPos*.

The Flush function

The *Flush* function is called at the end of each *Read*, *Readln*, *Write*, and *Writeln*. It can optionally flush the text-file buffer.

If *Mode* is *fmInput*, the *Flush* function can store zero in *BufPos* and *BufEnd* to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The *Close* function is called by the *CloseFile* standard procedure to close a text file associated with a device. (The *Reset*, *Rewrite*, and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutput*, then before calling *Close*, Object Pascal's file system calls the *InOut* function to ensure that all characters have been written to the device.

Chapter

17

Using null-terminated strings

Delphi supports a class of character strings call *null-terminated strings*. Null-terminated strings are widely used by the C and C++ programming languages, and by Windows itself. Using Delphi's null-terminated string support and the null-terminated string handling functions supplied by the *SysUtils* unit, you can easily interface with other languages and the Windows API.

What is a null-terminated string?

A null-terminated string consists of a sequence of non-null characters followed by a NULL (#0) character. Unlike Object Pascal strings, null-terminated strings have no separate length indicator. Instead, the first NULL character in a null-terminated string marks the end of the string.

Using null-terminated strings

Null-terminated strings are stored as arrays of characters with a zero-based integer index type; that is, an array of the form

```
array[0..X] of Char
```

where *X* is a positive nonzero integer. These arrays are called *zero-based character arrays*. Here are some examples of declarations of zero-based character arrays that can be used to store null-terminated strings:

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of Char;
```

The biggest difference between using Pascal strings and null-terminated strings is the extensive use of pointers in the manipulation of null-terminated strings.

Character pointers and string literals

A string literal is assignment-compatible with the *PChar* type. This means that a string literal can be assigned to a variable of type *PChar*. For example,

```
var
  P: PChar;
  ⋮
begin
  P := 'Hello world...';
end;
```

The effect of such an assignment is that the pointer points to an area of memory that contains a null-terminated copy of the string literal. This example accomplishes the same thing as the previous example:

```
const
  TempString: array[0..14] of Char = 'Hello world...'#0;
var
  P: PChar;
  ⋮
begin
  P := @TempString;
end;
```

You can use string literals as actual parameters in procedure and function calls when the corresponding formal parameter is of type *PChar*. For example, given a procedure with the declaration

```
procedure PrintStr(Str: PChar);
```

the following procedure calls are valid:

```
PrintStr('This is a test');
PrintStr(#10#13);
```

Just as it does with an assignment, the compiler generates a null-terminated copy of the string literal. The compiler passes a pointer to that memory area in the *Str* parameter of the *PrintStr* procedure.

Finally, you can initialize a typed constant of type *PChar* with a string constant. You can do this with structured types as well, such as arrays of *PChar* and records and objects with *PChar* fields.

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Character pointers and character arrays

A zero-based character array is *compatible* with the *PChar* type. This means that whenever a *PChar* is expected, you can use a zero-based character array instead.

When you use a character array in place of a *PChar* value, the compiler converts the character array to a pointer *constant* whose value corresponds to the address of the first element of the array. For example,

```
var
  A: array[0..63] of Char;
  P: PChar;
   :
begin
  A:= 'Assign';
  P := A;
  PrintStr(A);
  PrintStr(P);
end;
```

Because of the assignment statement, *P* points to the first element of *A*, so *PrintStr* is called twice with the same value.

You can initialize a typed constant of a zero-based character array type with a string literal that is shorter than the declared length of the array. The remaining characters are set to NULL (#0) and the array effectively contains a null-terminated string.

```
type
  TFileName = array[0..79] of Char;
var
  FileNameBuf: TFileName = 'TEST.PAS';
  FileNamePtr: PChar = FileNameBuf;
```

Character pointer indexing

Just as a zero-based character array is compatible with a character pointer, so can a character pointer be indexed as if it were a zero-based character array:

```
var
  A: array[0..63] of Char;
  P: PChar;
  Ch: Char;
   :
begin
  A:= 'Assign';
  P := A;
  Ch := A[5];
  Ch := P[5];
end;
```

Both of the last two statements assign *Ch* the value contained in the sixth character element of *A*.

When you index a character pointer, the index specifies a signed *offset* to add to the pointer before it is dereferenced. Therefore, *P[0]* is equivalent to *P^* and specifies the character pointed to by *P*. *P[1]* specifies the character right after the one pointed to by *P*, *P[2]* specifies the next character, and so on. Likewise, *P[–1]* specifies the character right before the one pointed to by *P*, *P[–2]* specifies the one before that, and so on.

The compiler performs no range checks when indexing a character pointer because it has no type information available to determine the maximum length of the null-terminated string pointed to by the character pointer. Your program must perform any such range checking.

The *StrUpper* function shown here illustrates the use of character pointer indexing to convert a null-terminated string to uppercase.

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

Notice that *StrUpper* is a function, not a procedure, and that it always returns the value that it was passed as a parameter. Because Object Pascal allows the result of a function call to be ignored, *StrUpper* can be treated as if it were a procedure:

```
StrUpper(A);
PrintStr(A);
```

However, as *StrUpper* always returns the *Dest* value it was passed, the preceding statements can be combined into one:

```
PrintStr(StrUpper(A));
```

Nesting calls to null-terminated string-handling functions can be very convenient when you want to indicate a certain interrelationship between a set of sequential string manipulations.

Note See "Character-pointer operators" on page 6-7 for information about *PChar* operations.

Null-terminated strings and standard procedures

Object Pascal's extended syntax allows the *Read*, *Readln*, *Str*, and *Val* standard procedures to be applied to zero-based character arrays, and allows the *Write*, *Writeln*, *Val*, *AssignFile*, and *Rename* standard procedures to be applied to both zero-based character arrays and character pointers.

Null-terminated string functions

The *SysUtils* unit provides a number of null-terminated string handling functions. The following table gives a brief description of each of these functions.

Table 17.1 Null-terminated string functions

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using *StrAlloc* or *StrNew*.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using *StrAlloc* or *StrNew*.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase.
StrMove	Moves a block of characters from one string to another.
StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to uppercase.

Mixing long strings and null-terminated strings

Object Pascal allows you to mix long strings and null-terminated strings in expressions and assignments. Long strings are described in "Long string types" on page 4-8. The rules that control mixing of long strings and null-terminated strings are outlined below.

- A null-terminated string is *assignment compatible* with a long string. In other words, a *PChar* value can be assigned to a variable of type **string**, or passed as a constant or value parameter of type **string**. For example, the assignment *S* := *P*, where *S* is a

string variable and *P* is a *PChar* expression, copies a null-terminated string into a long string.

- In an expression, if one operand of a binary operator is of type **string** and the other is of type *PChar*, the *PChar* operand is automatically converted to type **string**.
- A typecast of the form ***string(P)***, where *P* is a *PChar* expression, can be used to explicitly convert a null-terminated string to a long string. This is useful in situations where both operands of an operator are of type *PChar*, but you want to perform a string operation. For example,

```
S := string(P1) + string(P2);
```

  concatenates two null-terminated strings to form a resulting long string.
- A typecast of the form *PChar(S)*, where *S* is a long string expression, can be used to convert a long string to a null-terminated string.

Null-terminated wide character strings

The Windows operating system supports three types of character sets: *Single-byte character sets*, *double-byte character sets*, and the *Unicode character set*.

With a single-byte character set (SBCS), a character string is a sequence of bytes, and each byte represents an individual character. The ANSI character set used by most Western versions of Windows is a single-byte character set.

With a double-byte character set (DBCS), a character string is also a sequence of bytes, but unlike a single-byte character set, some characters are represented by one byte and others by two bytes. The first byte of a two byte character is called the *lead byte*. In general, the lower 128 characters of a double-byte character set map to the 7-bit ASCII character set, and lead bytes typically have ordinal values greater than or equal to 128. Double-byte character sets are widely used in Asia, where native character sets contain far more than 256 characters.

The Unicode character set is fundamentally different from single- and double-byte character sets in that each character is represented as a word (two bytes). A character string in the Unicode character set is a sequence of words, not bytes. Unicode characters are also known as *wide characters*, and Unicode strings are often referred to as *wide character strings*. With 65536 possible values for each character, Unicode can represent all the world's characters in modern computer use, including technical symbols and special characters used in publishing. The first 256 characters of the Unicode character set map to the ANSI character set.

Delphi supports single- and double-byte characters and character strings through the *AnsiChar*, *PAnsiChar*, and *AnsiString* fundamental types, and the *Char*, *PChar*, and **string** generic types. Wide characters and character strings are supported through the *WideChar* and *PWideChar* types, and the *WideString* type.

The null-terminated string handling features outlined earlier in this chapter also apply to the *PWideChar* type. This means that a string literal is assignment compatible with the *PWideChar* type (although the string literal can only contain wide characters

with ordinal values in the ANSI character range). Also, a zero-based wide character array of the form

```
array[0..X] of WideChar
```

is assignment compatible with the *PWideChar* type, and a value of type *PWideChar* can be indexed as if it were a zero-based wide character array. Furthermore, the descriptions of "Character-pointer operators" (+ and –) on page 6-7 also apply to wide character pointers.

Note When adding or subtracting integer offsets to and from wide character pointers, the offsets represent distances in wide characters, and are therefore automatically multiplied by two before being added to or subtracted from the pointers. Likewise, when subtracting one wide character pointer from another, the resulting integer is automatically divided by two to yield a distance in wide characters.

The *System* unit provides three functions, *WideCharToString*, *WideCharLenToString*, and *StringToWideChar*, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

Chapter

18

Memory issues

This chapter is about Object Pascal and memory. This chapter explains how Windows programs use memory, and it also describes the internal data formats used in Object Pascal.

Windows memory management

This section explains how Delphi applications use memory.

The memory manager

The memory manager manages all dynamic memory allocations and deallocations in a Delphi application. The *New*, *Dispose*, *GetMem*, *ReallocMem*, and *FreeMem* standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

Delphi's memory manager is highly optimized for applications that allocate large numbers of small to medium sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of *GlobalAlloc*, *LocalAlloc*, and private heap support in Windows, typically do not perform well in such scenarios, and would seriously slow down an application if they were used directly.

To ensure the best possible performance, the memory manager interfaces directly with the Win32 virtual memory API (the *VirtualAlloc* and *VirtualFree* functions). The memory manager *reserves* memory from the operating system in 1-MB sections of address space, and *commits* memory as required in 16-KB increments. Conversely, the memory manager *decommits* and *releases* unused memory in 16-KB and 1-MB sections. For smaller blocks, committed memory is further *sub-allocated*.

Memory manager blocks are always rounded upwards to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are

stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.

The memory manager maintains two status variables, *AllocMemCount* and *AllocMemSize*, which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging purposes.

The *System* unit provides two procedures, *GetMemoryManager* and *SetMemoryManager*, that allow applications to intercept low-level memory manager calls. Furthermore, the *System* unit provides a function called *GetHeapStatus* which returns a record containing detailed memory manager status information. For further information on these routines, see online help.

Internal data formats

The next several pages discuss the internal data formats of Object Pascal.

Integer types

The format selected to represent an integer-type variable depends on its minimum and maximum bounds:

- If both bounds are within the range –128..127 (*Shortint*), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (*Byte*), the variable is stored as an unsigned byte.
- If both bounds are within the range –32768..32767 (*Smallint*), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (*Word*), the variable is stored as an unsigned word.
- Otherwise, the variable is stored as a signed double word (*Longint*).

Character types

A *Char*, an *AnsiChar*, or a subrange of a *Char* type is stored as an unsigned byte. A *WideChar* is stored as an unsigned word.

Boolean types

A *Boolean* type is stored as a *Byte*, a *ByteBool* is stored as a *Byte*, a *WordBool* type is stored as a *Word*, and a *LongBool* is stored as a *Longint*.

A *Boolean* can assume the values 0 (*False*) and 1 (*True*). *ByteBool*, *WordBool*, and *LongBool* types can assume the values 0 (*False*) or nonzero (*True*).

Enumerated types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values, and if the type was declared in the **{$Z1}** state (the default). If an enumerated type has more than 256 values, or if the type was declared in the **{$Z2}** state, it is stored as an unsigned word. Finally, if an enumerated type is declared in the **{$Z4}** state, it is stored as an unsigned double-word.

Floating-point types

The floating-point types (*Real*, *Single*, *Double*, and *Extended*) store the binary representations of a sign (+or –), an *exponent*, and a *significand*. A represented number has the value

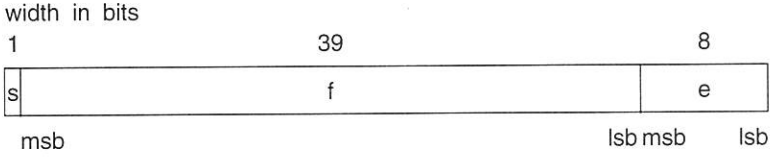
$$+/- \text{ significand} * 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary decimal point (that is, 0 <= significand < 2).

In the figures that follow, *msb* means most significant bit and *lsb* means least significant bit. The leftmost items are stored at the highest addresses. For example, for a *Real* value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real type

A 6-byte (48-bit) *Real* number is divided into three fields:



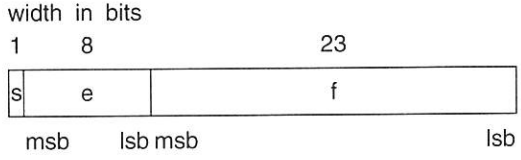
The value *v* of the number is determined by the following:

```
if 0 < e <= 255, then v = (-1)^s * 2^(e-129) * (1.f).
if e = 0, then v = 0.
```

The *Real* type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a *Real*, and NaNs and infinities produce an overflow error if an attempt is made to store them in a *Real*.

The Single type

A 4-byte (32-bit) *Single* number is divided into three fields:

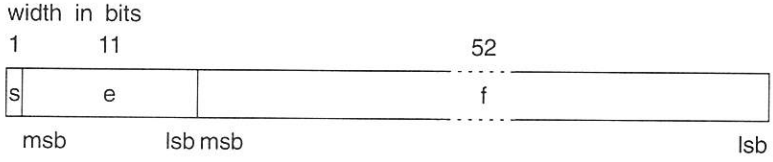


The value *v* of the number is determined by the following:

```
if 0 < e < 255, then v = (-1)^s * 2^(e-127) * (1.f).
if e = 0   and f <> 0, then v = (-1)^s * 2^(-126) * (0.f).
if e = 0   and f = 0, then v = (-1)^s * 0.
if e = 255 and f = 0, then v = (-1)^s * Inf.
if e = 255 and f <> 0, then v is a NaN.
```

The Double type

An 8-byte (64-bit) *Double* number is divided into three fields:

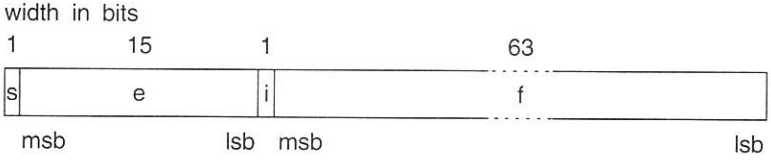


The value *v* of the number is determined by the following:

```
if 0 < e < 2047, then v = (-1)^s * 2^(e-1023) * (1.f).
if e = 0    and f <> 0, then v = (-1)^s * 2^(-1022) * (0.f).
if e = 0    and f = 0, then v = (-1)^s * 0.
if e = 2047 and f = 0, then v = (-1)^s * Inf.
if e = 2047 and f <> 0, then v is a NaN.
```

The Extended type

A 10-byte (80-bit) *Extended* number is divided into four fields:



The value *v* of the number is determined by the following:

```
if 0 <= e < 32767, then v = (-1)^s * 2^(e-16383) * (i.f).
if e = 32767 and f = 0, then v = (-1)^s * Inf.
if e = 32767 and f <> 0, then v is a NaN.
```

The Comp type

An 8-byte (64-bit) *Comp* number is stored as a signed 64-bit integer. The range of *Comp* values is –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$).

The Currency type

An 8-byte (64-bit) *Currency* number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places. The range of *Currency* values is –922,337,203,685,477.5808 to 922,337,203,685,477.5807.

Pointer types

A *Pointer* type is stored in 4 bytes as a 32-bit address. The pointer value **nil** is stored as zero.

Short string types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (**string**[255]).

Long string types

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a non-empty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. Table 18.1 shows the layout of a long string memory block.

Table 18.1 Long string dynamic memory layout

Offset	Contents
–8	32-bit reference count
–4	32-bit string length
0..*Length* – 1	Character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. The presence of the NULL character makes it possible to directly typecast a long string to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of –1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of –1.

Wide string types

A wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a non-empty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. Table 18.1 shows the layout of a wide string memory block.

Table 18.2 Wide string dynamic memory layout

Offset	Contents
–4	32-bit string length in bytes
0..*Length* – 1	Character string
Length	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. The presence of the NULL character makes it possible to directly typecast a wide string to a null-terminated string.

Set types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is calculated as

```
ByteSize = (Max div 8) - (Min div 8) + 1
```

where *Min* and *Max* are the lower and upper bounds of the base type of that set. The byte number of a specific element *E* is

```
ByteNumber = (E div 8) - (Min div 8)
```

and the bit number within that byte is

```
BitNumber = E mod 8
```

where *E* denotes the ordinal value of the element.

Array types

An array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

Record types

When a record type is declared in the {$A+} state (the default), and when the declaration does not include a packed modifier, the type is an *unpacked record type*, and the fields of the record are *aligned* for efficient access by the CPU. The alignment is controlled by the *alignment* of the type of each field. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, or 4, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. Table 18.3 lists the alignments for all data types.

Table 18.3 Type alignment masks

Type	Alignment
Ordinal types	Size of the type (1, 2, or 4)
Real types	2 for *Real* and *Extended*, 4 for all other real types
Short string types	1
Array types	Same as the element type of the array.
Record types	The largest alignment of the fields in the record
Set types	Size of the type if 1, 2, or 4, otherwise 1
All other types	4

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upwards to the byte boundary specified by the largest alignment of any of the fields.

When a record type is declared in the {$A–} state, or when the declaration includes the **packed** modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields.

File types

File types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```
type
  TFileRec = record
    Handle: Integer;
    Mode: Integer;
    RecSize: Cardinal;
    Private: array[1..28] of Byte;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
  end;
```

Text files occupy 460 bytes, which are laid out as follows:

```
type
```

```
TTextBuf = array[0..127] of Char;
TTextRec = record
  Handle: Integer;
  Mode: Integer;
  BufSize: Cardinal;
  BufPos: Cardinal;
  BufEnd: Cardinal;
  BufPtr: PChar;
  OpenFunc: Pointer;
  InOutFunc: Pointer;
  FlushFunc: Pointer;
  CloseFunc: Pointer;
  UserData: array[1..32] of Byte;
  Name: array[0..259] of Char;
  Buffer: TTextBuf;
end;
```

Handle contains the file's handle (when the file is open).

The *Mode* field can assume one of the following values:

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

fmClosed indicates that the file is closed. *fmInput* and *fmOutput* indicate that the file is a text file that has been reset (*fmInput*) or rewritten (*fmOutput*). *fmInOut* indicates that the file variable is a typed or an untyped file that has been reset or rewritten. Any other value indicates that the file variable hasn't been assigned (and thereby not initialized).

The *UserData* field is never accessed by Object Pascal and is free for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file. See "Text-file device drivers" on page 16-4, which provides information on that subject.

Procedural types

A global procedure pointer type is stored as a 32-bit pointer to the entry point of a procedure or function.

A method pointer type is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

Class types

A class type value is stored as a 32-bit pointer to an instance of the class. An instance of a class is also known as an object.

The internal data format of an object resembles that of a record. The fields of an object are stored in order of declaration as a contiguous sequences of variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table (VMT)* of the class. There is only one VMT per class (not one per instance), but two distinct class types never share a VMT, no matter how identical they appear to be. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Likewise, pointers to VMTs are automatically stored in class instances by the class type's constructor(s) and are never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointers, one per user-defined virtual method in the class type, in order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table, and the OLE Object Model used by Windows Object Linking and Embedding. At negative offsets, a VMT contains a number of fields that are internal to Object Pascal's implementation. These fields are listed here for informational purposes only. An application should use the methods defined in *TObject* to query this information, since the layout is likely to change in future implementations of Object Pascal.

Table 18.4 Virtual Method Table layout

Offset	Type	Description
–64	Pointer	Pointer to virtual method table (or **nil**).
–60	Pointer	Pointer to interface table (or **nil**).
–56	Pointer	Pointer to OLE Automation information table (or **nil**).
–52	Pointer	Pointer to instance initialization table (or **nil**).
–48	Pointer	Pointer to type information table (or **nil**).
–44	Pointer	Pointer to field definition table (or **nil**).
–40	Pointer	Pointer to method definition table (or **nil**).
–36	Pointer	Pointer to dynamic method table (or **nil**).
–32	Pointer	Pointer to short string containing class name.
–28	Cardinal	Instance size in bytes.
–24	Pointer	Pointer to a pointer to ancestor class (double indirection) (or **nil**).
–20	Pointer	Pointer to entrypoint of *SafecallException* method (or **nil**).
–16	Pointer	Entry point of *DefaultHandler* method.
–12	Pointer	Entry point of *NewInstance* method.
–8	Pointer	Entry point of *FreeInstance* method.
–4	Pointer	Entry point of *Destroy* destructor.
0	Pointer	Entry point of first user-defined virtual method.

Table 18.4 Virtual Method Table layout (continued)

Offset	Type	Description
4	Pointer	Entry point of second user-defined virtual method.
...	...	...

Class reference types

A class reference type value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

Variant types

A *Variant* is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the type code. The *System* unit defines the following constants and types for variants.

```
const
  varEmpty    = $0000;
  varNull     = $0001;
  varSmallint = $0002;
  varInteger  = $0003;
  varSingle   = $0004;
  varDouble   = $0005;
  varCurrency = $0006;
  varDate     = $0007;
  varOleStr   = $0008;
  varDispatch = $0009;
  varError    = $000A;
  varBoolean  = $000B;
  varVariant  = $000C;
  varUnknown  = $000D;
  varByte     = $0011;
  varString   = $0100;
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;

type
  TVarArrayBound = record
    ElementCount: Integer;
    LowBound: Integer;
  end;
  PVarArray = ^TVarArray;
  TVarArray = record
    DimCount: Word;
    Flags: Word;
    ElementSize: Integer;
    LockCount: Integer;
    Data: Pointer;
    Bounds: array[0..255] of TVarArrayBound;
  end;
```

```
PVarData = ^TVarData;
TVarData = record
  VType: Word;
  Reserved1, Reserved2, Reserved3: Word;
  case Integer of
    varSmallint: (VSmallint: Smallint);
    varInteger:  (VInteger: Integer);
    varSingle:   (VSingle: Single);
    varDouble:   (VDouble: Double);
    varCurrency: (VCurrency: Currency);
    varDate:     (VDate: Double);
    varOleStr:   (VOleStr: PWideChar);
    varDispatch: (VDispatch: Pointer);
    varError:    (VError: Integer);
    varBoolean:  (VBoolean: WordBool);
    varUnknown:  (VUnknown: Pointer);
    varByte:     (VByte: Byte);
    varString:   (VString: Pointer);
    varArray:    (VArray: PVarArray);
    varByRef:    (VPointer: Pointer);
end;
```

The *TVarData* type represents the internal structure of a *Variant* variable. The internal structure of a *Variant* is identical to the *Variant* type used by OLE and the Win32 APIs. The *TVarData* type can be used in typecasts of *Variant* variables to access the internals of a variable.

The *VType* field of a *TVarData* record contains the type code of the variant in the lower twelve bits (the bits defined by the *varTypeMask* constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.

The *Reserved1*, *Reserved2*, and *Reserved3* fields of a *TVarData* record are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field. If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.

If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines the specifics of the array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.

If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and *varArray* bits in the *VType* field.

Note that the *varString* type code is private to Delphi. Variants containing a *varString* value should never be passed to a non-Delphi function. Delphi's OLE Automation support will automatically convert *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

Chapter

19

Control issues

This chapter describes in detail the various ways that Delphi implements program control. Included are calling conventions and exit procedures.

Calling conventions

Parameters are transferred to procedures and functions via CPU registers and/or the stack, depending on the calling conventions of the procedures and functions. Delphi supports five calling conventions: **register, cdecl, pascal, stdcall and safecall**.

The **register** and **pascal** conventions pass parameters from left to right, that is the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl, stdcall and safecall** conventions pass parameters from right to left. For all conventions except **cdecl**, the procedure or function removes parameters from the stack upon returning. With the **cdecl** convention, the caller must remove parameters from the stack when the call returns. The **register** convention uses up to three CPU registers to pass parameters, whereas the other conventions always pass all parameters on the stack.

The default calling convention for methods of an interface is **register**. Interfaces intended to be used between modules, in particular modules written in different programming languages, should declare all methods with **stdcall**. The **safecall** calling convention must be used to implement methods of dual interfaces. For more information on interfaces, see "OLE automation Interface types" on page 13-14.

The calling conventions are summarized in the following table.

Table 19.1 Summary of calling conventions

Convention	Order	Cleanup	Registers
register	Left-to-right	Function	Yes
pascal	Left-to-right	Function	No

Table 19.1 Summary of calling conventions

Convention	Order	Cleanup	Registers
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Function	No
safecall	Right-to-left	Function	No

Parameters

The method used to pass a particular parameter depends on whether the parameter is a value, constant, or variable parameter.

Variable parameters are always passed by reference, that is as 32-bit pointers that point to the actual storage location.

Value and constant parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal type parameter is passed as an 8-bit, 16-bit, or 32-bit value, using the same format as a variable of the corresponding type.
- A real type parameter is always passed on the stack. A *Single* parameter occupies 4 bytes, and a *Double*, *Comp*, or *Currency* parameter occupies 8 bytes. A *Real* occupies 8 bytes, with the *Real* value stored in the lower 6 bytes. An *Extended* occupies 12 bytes, with the *Extended* value stored in the lower 10 bytes.
- A short string type parameter is passed as a 32-bit pointer to a short string.
- A long string type parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value **nil** is passed for an empty long string.
- A pointer type, class type, class reference type, or global procedure pointer type parameter is passed as a 32-bit pointer.
- A method pointer type is always passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- A *Variant* parameter is passed as a 32-bit pointer to a *Variant* value with the **register** and **pascal** conventions.
- Arrays, records, and sets of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32-bit values. Other arrays, records, and sets are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack with the **cdecl**, **stdcall** and **safecall** conventions. The size of a record passed this way is always rounded upwards to the nearest double-word boundary.
- An open-array parameter is passed as two 32-bit values. The first 32-bit value is a pointer to the array data. The second 32-bit value is the number of elements in the array less one.

When passing parameters on the stack, each parameter always occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even

though the parameter only occupies a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

With the **pascal**, **cdecl**, **stdcall** and **safecall** conventions, all parameters are passed on the stack. For the **pascal** convention, parameters are pushed in their order of declaration (left-to-right), so that the first parameter ends up at the highest address and the last parameter ends up at the lowest address. For the **cdecl**, **stdcall and safecall** conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

With the **register** convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in their order of declaration (same as the **pascal** convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real types and method pointer types do not qualify as register parameters, but all other parameter types do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in their order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to *Test* will pass *A* in EAX as a 32-bit integer, *B* in EDX as a pointer to a *Char*, and *D* in ECX as a pointer to a long string memory block, and will push *C* onto the stack as two double-words and push *E* onto the stack as a 32-bit pointer, in that order.

Function results

The following conventions are used for returning function result values:

- Ordinal type function results are returned in a CPU register: Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real type function results are returned in the floating-point co-processor's top-of-stack register (ST(0)). For function results of type *Currency*, the value in ST(0) is scaled by 10000. For example, the *Currency* value 1.234 is returned in ST(0) as 12340.
- For a string, method pointer, or *Variant* type result, the effects are the same as if the function result was declared as an additional **var** parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Pointer type, class type, class reference type, and global procedure pointer type function results are returned in EAX.
- For array, record, and set type function results, if the value occupies one byte it is returned in AL, if the value occupies two bytes it is returned in AX, and if the value occupies four bytes it is returned in EAX. Otherwise, an additional **var** parameter is passed to the function, following the declared parameters, in which the function returns the result.

Method calling conventions

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter, *Self*, which is a reference to the instance or class for which the method is invoked. The *Self* parameter is always passed as a 32-bit pointer.

- With the **register** convention, the *Self* parameter behaves as if it was declared *before* all other parameters. It is therefore always passed in the EAX register.
- With the **pascal** convention, the *Self* parameter behaves as if it was declared *after* all other parameters, including the additional **var** parameter that might be passed for a function result. It is therefore always pushed last, ending up at a lower address than all other parameters.
- For the **cdecl**, **stdcall** and **safecall** conventions, the *Self* parameter behaves as if it was declared *before* all other parameters, but *after* the additional **var** parameter that might be passed for a function result. It is therefore always pushed last, but before the additional **var** parameter for the function result (if any).

Constructors and destructors

Constructors and destructors use the same calling conventions as other methods, except that an additional *Boolean* flag parameter is passed to indicate the context of the constructor or destructor call.

A value of *False* in the flag parameter of a constructor call indicates that the constructor was invoked through an instance or using the **inherited** keyword. In this case, the constructor behaves like an ordinary method.

A value of *True* in the flag parameter of a constructor call indicates that the constructor was invoked through a call reference. In this case, the constructor creates an instance of the class given by *Self*, and returns a reference to the newly created object in EAX.

A value of *False* in the flag parameter of a destructor call indicates that the destructor was invoked using the **inherited** keyword. In this case, the destructor behaves like an ordinary method.

A value of *True* in the flag parameter of a destructor call indicates that the destructor was invoked through an instance. In this case, the destructor deallocates the instance given by *Self* just before returning.

For all calling conventions, the *Boolean* flag parameter behaves as if it was declared before all other parameters. Thus, with the **register** convention, the flag is always passed in the DL register, with the **pascal** convention, the flag is always pushed before all other parameters, and with the **cdecl**, **stdcall** and **safecall** conventions, the flag is always pushed right before the *Self* parameter.

Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can freely modify the EAX, EDX, and ECX registers. In addition, procedures and functions are always invoked with the assumption that the CPU's *direction flag is* cleared (corresponding to a CLD instruction), and must always return with the direction flag cleared.

Exit procedures

By installing an exit procedure, you can gain control over a program's termination process. This is useful when you want to make sure specific actions are carried out before a program terminates; a typical example is updating and closing files.

Note Exit procedures can only be used when generating an .EXE file. When using packages, exit behavior must be implemented in a finalization section. It is recommended that finalization sections be used for all exit behavior. See "The finalization part" on page 12-4.

All exit procedures are called before any finalization sections are executed.

The *ExitProc* pointer variable allows you to install an exit procedure. The exit procedure is always called as a part of a program's termination, whether it's a normal termination, a termination through a call to *Halt*, or a termination due to a runtime error.

An exit procedure takes no parameters.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. This chain makes it possible for units as well as programs to install exit procedures. Some units install an exit procedure as part of their initialization code and then rely on that specific procedure to be called to clean up after the unit. Closing files is such an example. The procedures on the exit chain are executed in reverse order of installation. This ensures that the exit code of one unit isn't executed before the exit code of any units that depend upon it.

To keep the exit chain intact, you must save the current contents of *ExitProc* before changing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of *ExitProc*. The following code demonstrates a skeleton method of implementing an exit procedure:

```
var
  ExitSave: Pointer;

procedure MyExit;
begin
  ExitProc := ExitSave;          { Always restore old vector first }
  :
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
```

```
  :
end.
```

On entry, the code saves the contents of *ExitProc* in *ExitSave*, and then installs the *MyExit* exit procedure. After having been called as part of the termination process, the first thing *MyExit* does is reinstall the previous exit procedure

The termination routine in the runtime library keeps calling exit procedures until *ExitProc* becomes **nil**. To avoid infinite loops, *ExitProc* is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to *ExitProc*. If an error occurs in an exit procedure, it won't be called again.

An exit procedure can learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable.

In case of normal termination, *ExitCode* is zero and *ErrorAddr* is **nil**. In case of termination through a call to *Halt*, *ExitCode* contains the value passed to *Halt*, and *ErrorAddr* is **nil**. Finally, in case of termination due to a runtime error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the statement in error.

The last exit procedure (the one installed by the runtime library) closes the *Input* and *Output* files. If *ErrorAddr* is not **nil**, it outputs a runtime error message.

If you wish to present runtime error messages yourself, install an exit procedure that examines *ErrorAddr* and outputs a message if it's not **nil**. In addition, before returning, make sure to set *ErrorAddr* to **nil**, so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to Windows, passing the value stored in *ExitCode* as a return code.

Chapter

20

The built-in assembler

Delphi's built-in assembler allows you to write Intel assembler code directly inside your Object Pascal programs. Of course, you can still link in .OBJ files that contain **external** procedures and functions when you want to mix Object Pascal and assembler.

The built-in assembler implements a large subset of the syntax supported by Turbo Assembler and Microsoft's Macro Assembler. The built-in assembler supports all 8086/8087 and 80386/80387 opcodes, and all but a few of Turbo Assembler's expression operators.

Except for DB, DW, and DD (define byte, word, and double word), none of Turbo Assembler's directives, such as EQU, PROC, STRUC, SEGMENT, and MACRO, are supported by the built-in assembler. Operations implemented through Turbo Assembler directives, however, are largely matched by corresponding Delphi constructs. For example, most EQU directives correspond to **const**, **var**, and **type** declarations in Delphi, the PROC directive corresponds to **procedure** and **function** declarations, and the STRUC directive corresponds to Delphi **record** types. In fact, Delphi's built-in assembler can be thought of as an assembler language compiler that uses Object Pascal syntax for all declarations.

The asm statement

The built-in assembler is accessed through **asm** statements. This is the syntax of an **asm** statement:

```
asm AsmStatement [ Separator AsmStatement ] end
```

where *AsmStatement* is an assembler statement, and *Separator* is a semicolon, a new-line, or a Object Pascal comment.

Multiple assembler statements can be placed on one line if they are separated by semicolons. A semicolon isn't required between two assembler statements if the statements are on separate lines. A semicolon doesn't indicate that the rest of the line

is a comment—comments must be written in Object Pascal style using { and } or (*and *).

Register use

In general, the rules of register use in an **asm** statement are the same as those of an **external** procedure or function. An **asm** statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an **asm** statement, BP points to the current stack frame, SP points to the top of the stack, SS contains the segment address of the stack segment, and DS contains the segment address of the data segment. Except for EDI, ESI, ESP, EBP, and EBX, an **asm** statement can assume nothing about register contents on entry to the statement.

Assembler statement syntax

This is the syntax of an assembler statement:

```
[ Label ":" ] < Prefix > [ Opcode [ Operand < "," Operand > ] ]
```

Label is a label identifier, *Prefix* is an assembler prefix opcode (operation code), *Opcode* is an assembler instruction opcode or directive, and *Operand* is an assembler expression.

Comments are allowed between assembler statements, but not within them. For example, this is allowed:

```
asm
  MOV AX,1 {Initial value}
  MOV CX,100 {Count}
end;
```

but this is an error:

```
asm
  MOV {Initial value} AX,1;
  MOV CX, {Count} 100
end;
```

Labels

Labels are defined in assembler as they are in Object Pascal—by writing a label identifier and a colon before a statement. There is no limit to a label length, except only the first 32 characters of an identifier are significant in the built-in assembler. And as they are in Object Pascal, labels defined in assembler must be declared in a **label** declaration part in the block containing the **asm** statement. There is one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). Because an at-sign can't be part of a Object Pascal identifier, such local labels are automatically restricted to use within **asm** statements. A local label is known only within the **asm** statement that defines it

(that is, the scope of a local label extends from the **asm** keyword to the **end** keyword of the **asm** statement that contains it).

Unlike a normal label, a local label doesn't have to be declared in a **label** declaration part before it's used.

The exact composition of a local label identifier is an at-sign (@) followed by one or more letters (*A*.. *Z*), digits (0..9), underscores (_), or at-signs. As with all labels, the identifier is followed by a colon (:).

Instruction opcodes

The built-in assembler supports all 8086/8087 and 80386/80387 instruction opcodes.

For a complete description of each instruction, refer to your 80x86 and 80x87 reference manuals.

RET instruction sizing

The RET instruction opcode always generates a near return.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and all conditional jump instructions, when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one byte opcode followed by a one byte displacement) if the distance to the target label is within –128 to 127 bytes; otherwise a near jump (one byte opcode followed by a two byte displacement) is generated.

For a conditional jump instruction, a short jump (1 byte opcode followed by a 1 byte displacement) is generated if the distance to the target label is within –128 to 127 bytes; otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (5 bytes in total). For example, the assembler statement

```
JC      Stop
```

where *Stop* isn't within reach of a short jump is converted to a machine code sequence that corresponds to this:

```
JNC     Skip
JMP     Stop
Skip:
```

Jumps to the entry points of procedures and functions are always near.

Assembler directives

Delphi's built-in assembler supports three assembler directives: DB (define byte), DW (define word), and DD (define double word). They each generate data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between –128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between –32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer, that is, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between –2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer, that is, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembler statements. To generate uninitialized or initialized data in the data segment, you should use Object Pascal **var** or **const** declarations.

Some examples of DB, DW, and DD directives follow:

```
asm
  DB      0FFH                        { One byte }
  DB      0,99                        { Two bytes }
  DB      'A'                         { Ord('A') }
  DB      'Hello world...',0DH,0AH    { String followed by CR/LF }
  DB      12,"Delphi"                 { Object Pascal style string }
  DW      0FFFFH                      { One word }
  DW      0,9999                      { Two words }
  DW      'A'                         { Same as DB 'A',0 }
  DW      'BA'                        { Same as DB 'A','B' }
  DW      MyVar                       { Offset of MyVar }
  DW      MyProc                      { Offset of MyProc }
  DD      0FFFFFFFFH                  { One double-word }
  DD      0,999999999                 { Two double-words }
  DD      'A'                         { Same as DB 'A',0,0,0 }
  DD      'DCBA'                      { Same as DB 'A','B','C','D' }
  DD      MyVar                       { Pointer to MyVar }
  DD      MyProc                      { Pointer to MyProc }
end;
```

In Turbo Assembler, when an identifier precedes a DB, DW, or DD directive, it causes the declaration of a byte, word, or double word sized variable at the location of the directive. For example, Turbo Assembler allows the following:

```
ByteVar     DB      ?
WordVar     DW      ?
```

```
IntVar      DD      ?
  :
            MOV     AL,ByteVar
            MOV     BX,WordVar
            MOV     ECX,IntVar
```

The built-in assembler doesn't support such variable declarations. In Delphi, the only kind of symbol that can be defined in an built-in assembler statement is a label. All variables must be declared using Object Pascal syntax, and the preceding construct corresponds to this:

```
var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
  :
asm
  MOV     AL,ByteVar
  MOV     BX,WordVar
  MOV     ECX,IntVar
end;
```

Operands

Built-in assembler operands are expressions that consist of a combination of constants, registers, symbols, and operators. Although built-in assembler expressions are built using the same basic principles as Object Pascal expressions, there are a number of important differences, as will be explained later in this chapter.

Within operands, the reserved words in Table 19.1 have a predefined meaning to the built-in assembler:

Table 20.1 Built-in assembler reserved words

AH	BX	DI	EBX	ESP	PTR	SS
AL	BYTE	DL	ECX	HIGH	QWORD	ST
AND	CH	DS	EDI	LOW	SHL	TBYTE
AX	CL	DWORD	EDX	MOD	SHR	TYPE
BH	CS	DX	EID	NOT	SI	WORD
BL	CX	EAX	ES	OFFSET	SP	XOR
BP	DH	EBP	ESI	OR		

The reserved words always take precedence over user-defined identifiers. For example, the code fragment,

```
var
  ch: Char;
  :
asm
  MOV     CH, 1
end;
```

loads 1 into the CH register, *not* into the *CH* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) identifier override operator:

```
asm
  MOV     &ch, 1
end;
```

It's strongly suggested that you avoid user-defined identifiers with the same names as built-in assembler reserved words, because such name confusion can easily lead to obscure and hard-to-find bugs.

Expressions

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants.

Built-in assembler expressions are built from *expression elements* and *operators*, and each expression has an associated *expression class* and *expression type*. These concepts are explained in the following sections.

Differences between Object Pascal and Assembler expressions

The most important difference between Object Pascal expressions and built-in assembler expressions is that all built-in assembler expressions must resolve to a *constant value*, a value that can be computed at compile time. For example, given these declarations:

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid built-in assembler statement:

```
asm
  MOV     Z,X+Y
end;
```

Because both *X* and *Y* are constants, the expression *X* + *Y* is merely a more convenient way of writing the constant 30, and the resulting instruction becomes a move immediate of the value 30 into the word-sized variable *Z*. But if you change *X* and *Y* to be variables,

```
var
  X, Y: Integer;
```

the built-in assembler can no longer compute the value of *X* + *Y* at compile time. The correct built-in assembler construct to move the sum of *X* and *Y* into *Z* is this:

```
asm
  MOV     EAX,X
  ADD     EAX,Y
```

```
  MOV     Z,EAX
end;
```

Another important difference between Object Pascal and built-in assembler expressions is the way variables are interpreted. In a Object Pascal expression, a reference to a variable is interpreted as the *contents* of the variable, but in an built-in assembler expression, a variable reference denotes the *address* of the variable. For example, in Object Pascal, the expression *X* + 4, where *X* is a variable, means the contents of *X plus* 4, whereas in the built-in assembler, it means the contents of the word at an address four bytes higher than the address of *X*. So, even though you're allowed to write

```
asm
  MOV     EAX,X+4
end;
```

the code doesn't load the value of *X* plus 4 into AX, but it loads the value of a word stored four bytes beyond *X* instead. The correct way to add 4 to the contents of *X* is like this:

```
asm
  MOV     EAX,X
  ADD     EAX,4
end;
```

Expression elements

The basic elements of an expression are *constants*, *registers*, and *symbols*.

Constants

The built-in assembler supports two types of constants: *numeric constants* and *string constants*.

Numeric constants

Numeric constants must be integers, and their values must be between –2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal (base 10) notation, but the built-in assembler supports binary (base 2), octal (base 8), and hexadecimal (base 16) notations as well. Binary notation is selected by writing a *B* after the number, octal notation is selected by writing a letter *O* after the number, and hexadecimal notation is selected by writing an *H* after the number or a $ before the number.

The *B*, *O*, and *H* suffixes aren't supported in Object Pascal expressions. Object Pascal expressions allow only decimal notation (the default) and hexadecimal notation (using a $ prefix).

Numeric constants must start with one of the digits 0 through 9 or a $ character; therefore when you write a hexadecimal constant using the *H* suffix, an extra zero in front of the number is required if the first significant digit is one of the hexadecimal digits *A* through *F*. For example, 0BAD4H and $BAD4 are hexadecimal constants, but BAD4H is an identifier because it starts with a letter and not a digit.

String constants

String constants must be enclosed in single or double quotes. Two consecutive quotes of the same type as the enclosing quotes count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
"That's all folks"
'"That''s all folks," he said.'
'100'
'"'
"'"
```

Notice in the fourth string the use of two consecutive single quotes to denote one single quote character.

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters, and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the leftmost (first) character(s) are assumed to be zero. Table 19.2 presents some examples of string constants and their corresponding numeric values:

Table 20.2 String examples and their values

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers:

Table 20.3 CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it's called a *register operand*. All registers can be used as register operands. In addition, some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registers, such as [EAX+ECX], [ESP], and [ESP+EAX+5].

The 16-bit segment registers (ES, CS, SS, and DS) are supported, but segments are not available in 32-bit code.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(x), where x is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Object Pascal symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the following special symbol:

@Result

The *@Result* symbol represents the function result variable within the statement part of a function. For example, in this function,

```
function Sum(X, Y: Integer): Integer;
begin
  Sum := X + Y;
end;
```

the statement that assigns a function result value to *Sum* uses the *@Result* variable if it is written in built-in assembler:

```
function Sum(X, Y: Integer): Integer; pascal;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

The following symbols can't be used in built-in assembler expressions:

- Standard procedures and functions (for example, *WriteLn*, *Chr*)
- The *Mem*, *MemW*, *MemL*, *Port*, and *PortW* special arrays

- String, floating-point, and set constants
- Labels that aren't declared in the current block
- The *@Result* symbol outside a function

Table 20.4 summarizes the value, class, and type of the different kinds of symbols that can be used in built-in assembler expressions. (Expression classes and types are described in a following section.)

Table 20.4 Values, classes, and types of symbols

Symbol	Value	Class	Type
Label	Address of label	Memory	SHORT
Constant	Value of constant	Immediate	0
Type	0	Memory	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory	Size of type
Procedure	Address of procedure	Memory	NEAR
Function	Address of function	Memory	NEAR
Unit	0	Immediate	0
@Code	Code segment address	Memory	0FFF0H
@Data	Data segment address	Memory	0FFF0H
@Result	Result var offset	Memory	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given these declarations,

```
procedure Test;
var
  Count: Integer;
```

the instruction

```
asm
  MOV     EAX,Count
end;
```

assembles into MOV EAX,[EBP–4].

The built-in assembler always treats a **var** parameter as a 32-bit pointer, and the size of a **var** parameter is always 4 (the size of a 32-bit pointer). In Object Pascal, the syntax for accessing a **var** parameter and a value parameter is the same—this isn't the case in code you write for the built-in assembler. Because **var** parameters are really pointers, you have to treat them as such. So, to access the contents of a **var** parameter, you first have to load the 32-bit pointer and then access the location it points to. For example, if the X and Y parameters of the above function *Sum* were **var** parameters, the code would look like this:

```
function Sum(var X, Y: Integer): Integer; pascal;
begin
  asm
    LES     BX,X
    MOV     AX,ES:[BX]
    LES     BX,Y
    ADD     AX,ES:[BX]
    MOV     @Result,AX
  end;
end;
```

Some symbols, such as record types and variables, have a scope that can be accessed using the period (.) structure member selector operator. For example, given these declarations,

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

the following constructs can be used to access fields in the *P* and *R* variables:

```
asm
  MOV     AX,P.X
  MOV     DX,P.Y
  MOV     CX,R.A.X
  MOV     BX,R.B.Y
end;
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of ES:[DI+4] into AX:

```
asm
  MOV     AX,(TRect PTR ES:[DI]).B.X
  MOV     AX,TRect(ES:[DI]).B.X
  MOV     AX,ES:TRect[DI].B.X
  MOV     AX,TRect[ES:DI].B.X
  MOV     AX,ES:[DI].TRect.B.X
end;
```

A scope is provided by type, field, and variable symbols of a record or object type. In addition, a unit identifier opens the scope of a particular unit, just like a fully qualified identifier in Object Pascal.

Expression classes

The built-in assembler divides expressions into three classes: *registers*, *memory references*, and *immediate values*.

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references; Object Pascal's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values; this group includes Object Pascal's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;
  :
asm
  MOV     EAX,Start           { MOV EAX,xxxx }
  MOV     EBX,Count           { MOV EBX,[xxxx] }
  MOV     ECX,[Start]         { MOV ECX,[xxxx] }
  MOV     EDX,OFFSET Count    { MOV EDX,xxxx }
end;
```

Because *Start* is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as *Count* is a memory reference. In the third MOV, the square brackets operator is used to convert *Start* into a memory reference (in this case, the word at offset 10 in the data segment), and in the fourth MOV, the OFFSET operator is used to convert *Count* into an immediate value (the offset of *Count* in the data segment).

As you can see, the square brackets and the OFFSET operators complement each other. In terms of the resulting machine code, the following **asm** statement is identical to the first two lines of the previous **asm** statement:

```
asm
  MOV     AX,OFFSET [Start]
  MOV     BX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either *relocatable expressions* or *absolute expressions*. A relocatable expression denotes a value that requires *relocation* at link time, and an absolute expression denotes a value that requires no such relocation. Typically, an expression that refers to a label, variable, procedure, or function is relocatable, and an expression that operates solely on constants is absolute.

Relocation is the process by which the linker assigns absolute addresses to symbols. At compile time, the compiler doesn't know the final address of a label, variable, procedure, or function; it doesn't become known until link time, when the linker assigns a specific absolute address to the symbol.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression types

Every built-in assembler expression has an associated type—or more correctly, an associated size, because the built-in assembler regards the type of an expression simply as the size of its memory location. For example, the type (size) of an *Integer* variable is two, because it occupies 2 bytes.

The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  :
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

the built-in assembler checks that the size of *QuitFlag* is one (a byte), and that the size of *OutBufPtr* is two (a word). An error results if the type check fails; for example, this isn't allowed:

```
asm
  MOV     DL,OutBufPtr
end;
```

The problem is DL is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
asm
  MOV     DL,BYTE PTR OutBufPtr
  MOV     DL,Byte(OutBufPtr)
  MOV     DL,OutBufPtr.Byte
end;
```

These MOV instructions all refer to the first (least significant) byte of the *OutBufPtr* variable.

In some cases, a memory reference is untyped; that is, it has no associated type. One example is an immediate value enclosed in square brackets:

```
asm
  MOV     AL,[100H]
  MOV     BX,[100H]
end;
```

The built-in assembler permits both of these instructions, because the expression [100H] has no associated type—it just means "the contents of address 100H in the data segment," and the type can be determined from the first operand (byte for AL,

word for BX). In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast:

```
asm
  INC     BYTE PTR [100H]
  IMUL    WORD PTR [100H]
end;
```

Table 19.5 summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Object Pascal types.

Table 20.5 Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Expression operators

The built-in assembler provides a variety of operators, divided into 12 classes of precedence. Table 19.6 lists the built-in assembler's expression operators in decreasing order of precedence.

Built-in assembler operator precedence is different from Object Pascal. For example, in a built-in assembler expression, the AND operator has lower precedence than the plus (+) and minus (–) operators, whereas in an Object Pascal expression, it has higher precedence.

Table 20.6 Summary of built-in asssembler expression operators

Operator(s)	Comments
&	Identifier override operator
(), [],HIGH, LOW	Structure member selector
+, –	Unary operators
:	Segment override operator
OFFSET, SEG, TYPE, PTR, *, /, MOD, SHL, SHR,+, –	Binary addition / subtraction operators
NOT, AND, OR, XOR	Bitwise operators

Table 20.7 Definitions of built-in assembler expression operators

Operator	Description
&	**Identifier override**. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	**Subexpression**. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can optionally precede the expression within the parentheses; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference**. The expression within brackets is evaluated completely prior to being treated as a single expression element. The expression within brackets can be combined with the BX, BP, SI, or DI registers using the plus (+) operator, to indicate CPU register indexing. Another expression can optionally precede the expression within the brackets; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector**. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus**. Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
–	**Unary minus**. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
:	**Segment override**. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction will be prefixed by an appropriate segment override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator**. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	**Multiplication**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Table 20.7 Definitions of built-in assembler expression operators (continued)

Operator	Description
+	**Addition**. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions are memory references, the result is also a memory reference.
–	**Subtraction**. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembler procedures and functions

So far, every **asm**... **end** construct you've seen has been a statement within a normal **begin**... **end** statement part. Delphi's assembler directive allows you to write complete procedures and functions in built-in assembler, without the need for a **begin**... **end** statement part. Here's an example of an assembler function:

```
function LongMul(X, Y: Integer): Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

The assembler directive causes Delphi to perform a number of code generation optimizations:

- The compiler doesn't generate code to copy value parameters into local variables. This affects all string-type value parameters, and other value parameters whose size isn't 1, 2, or 4 bytes. Within the procedure or function, such parameters must be treated as if they were **var** parameters.
- The compiler doesn't allocate a function result variable, and a reference to the *@Result* symbol is an error. String functions, variants and interfaces, however, are an exception to this rule—they always have a *@Result* pointer that is allocated by the caller.
- The compiler generates no stack frame for procedures and functions that aren't nested and have no parameters and no local variables.
- The automatically generated entry and exit code for an assembler procedure or function looks like this:

```
PUSH    EBP             ;Present if Locals <> 0 or Params <> 0
MOV     EBP,ESP          ;Present if Locals <> 0 or Params <> 0
SUB     ESP,Locals      ;Present if Locals <> 0
 :
MOV     ESP,EBP          ;Present if Locals <> 0
POP     EBP             ;Present if Locals <> 0 or Params <> 0
RET     Params          ;Always present
```

If locals include variants, long strings or interfaces, they are initialized to zero, but not finalized.

- *Locals* is the size of the local variables, and *Params* is the size of the parameters. If both *Locals* and *Params* are zero, there is no entry code, and the exit code consists simply of a RET instruction.

Functions using the **assembler** directive must return their results as follows:

- Ordinal-type function results (integer, Boolean, enumerated types, and *Char*) are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real-type function results (type *Single*, *Double*, *Extended*, *Comp*, and *Currency*) are returned in ST(0) on the coprocessor's register stack. Note that *Currency* values are scaled by 10000.
- Pointer-type function results, including long strings, are returned in EAX.
- Short string-type function results and variants are returned in the temporary location pointed to by the *@Result* function result symbol.

Index

Symbols

(pound) character in code 2-5
$ (dollar sign) prefix 2-4, 2-6
$0EEDFACE exception
  code 14-8
& operator 20-6
(* symbol 2-2
(. symbol 2-2
* operator 6-4
    set expressions 6-7
    variant expressions 4-25
*) symbol 2-2
+ operator
    binary expressions 6-4
    character pointers 6-7
    concatenation 6-6
    set expressions 6-7
    unary expressions 6-4
    variant expressions 4-25
. (dot) operator 20-11, 20-15
.) symbol 2-2
/ operator 6-4
    variant expressions 4-25
:= (assignment) symbol 4-30
; (semicolon) in code 7-4
< operator 6-8
    variant expressions 4-25
<= operator 6-8
    variant expressions 4-25
<> operator 6-8
    variant expressions 4-25
= operator 6-8
    variant expressions 4-25
> operator 6-8
    variant expressions 4-25
>= operator 6-8
@ (at-sign) in assembler
  code 20-2
@ operator 6-10
    pointers and 4-16, 5-5
    procedural types 6-13
    properties and 10-20
    variable typcasting and 5-6
@@ operator 6-14
[ ] (brackets) in expressions 6-11
    built-in assembler 20-9
[symbol 2-2
] symbol 2-2
^ (pointer) symbol 5-5
    error conditions 4-16
    field designators and 5-3
    object references 10-3
{ symbol 2-2
} symbol 2-2
– operator
    binary expressions 6-4
    character pointers 6-7
    set expressions 6-7
    unary expressions 6-4
    variant expressions 4-25
' (apostrophe) in code 2-5
    as literal character 2-5

Numerics

16-bit applications
    integer types 4-3
32-bit applications
    integer types 4-3
32-bit pointers 20-10
32-bit registers 20-9
80386/80387 processors 20-1
8086/8087 processors 20-1
    register stack 20-9

A

$A compiler directive 18-7
Abs function 3-2
absolute addresses 20-12
absolute expressions
  (assembler) 20-12
abstract methods 10-10
abstract reserved word 10-10
access specifiers 10-21, 10-22
    index specifiers and 10-23
    inherited properties
      and 10-25
    interface objects 13-5
    property overrides
      and 10-25
accessing files 16-2
accessing properties 10-20,
  10-21, 10-22
    multiple 10-23
ActiveX applications
    string types 4-8
actual parameters 7-2
adding units to programs 12-2
addition 6-4
Addr function 3-2
_AddRef method 13-4, 13-7,
  13-11
addresses 5-7, 19-3
    absolute 20-12
    dynamic variables 4-16
    factors of 6-10
    functions 4-18
    global procedures 4-18
    pointer variables and 5-5
    procedural variables 6-14
    stack 20-2
address-of (@) operator 6-10
    pointers and 4-16, 5-5
    procedural types 6-13
    properties and 10-20
    variable typcasting and 5-6
alignment 18-7
    structured types and 4-12
alignment masks 18-7
allocating memory 5-2, 18-1
    dynamic variables 4-16
    objects and 10-3
AllocMemCount variable 18-2
AllocMemSize variable 18-2
ancestors 10-4, 10-5
    inherited fields 18-9
    interface objects 13-3, 13-7
    message handlers 10-17
and operator
    bitwise operations 6-5
    Boolean operations 6-6
    variant expressions 4-25
ANSI characters 4-4, 4-8, 17-6
    conversions 4-24
AnsiChar type 4-4
    internal formats 18-2
AnsiString type 4-8, 4-11
    pointers 4-17
apostrophe (') in code 2-5
    as literal character 2-5
Append procedure 16-1, 16-2
    device drivers 16-5
applications
    console 16-3
    destroying exceptions 11-2,
      11-8
    exporting functions and 14-7
    integer types and 4-4
    multi-language
      programming 13-5, 14-1
    multi-threaded 14-9

running 15-1
runtime type
information 10-6, 10-24
checking 10-15
sharing resources 14-1
stack size 5-2
arithmetic expressions 6-4 to 6-5
arithmetic operators 6-4
built-in assembler 20-15
precedence 6-1
precison 4-4, 4-7
array properties 10-21 to 10-23
accessing 10-22
interface objects 13-5
published directive and 10-6
array types 4-12 to 4-13
constants as 5-8
return values 19-3
storage 18-6
arrays 4-12
accessing elements 4-13, 4-27
arrays of 4-12
as parameters 19-2
automation objects and 4-30
character 5-8, 17-3, 17-7
zero-based 4-13, 17-1
initializing 5-9, 17-3
interface objects and 13-14
multidimensional 4-12, 4-27, 5-9
indexing 5-4
null-terminated strings 17-3
objects 5-3
passing to procedures and functions 9-9
qualified names and 5-3
records 5-3
references 5-3
resizing 4-27
storage 4-17
strings 4-8, 4-9
compatibility 2-6
variables and 5-3
variable-sized 4-13, 4-27, 9-9
variants and 4-20, 4-22, 4-26 to 4-28
wide character 17-7
as operator 10-16
class-reference types and 10-26
interface objects 13-13
ASCII characters 2-1, 17-6
identifiers and 2-3
strings and 2-5
asm reserved word 20-1, 20-2
asm statements 20-1 to 20-2
local labels 20-2
assembler 20-1
See also assembly language
accessing 20-1
code generation 20-16
expression types 20-6
opcodes 20-3
overriding identifiers 20-15
reserved symbols 20-9
types 20-14
reserved words 20-5
typecasting 20-14, 20-15
assembler directives 20-4 to 20-5
compatibility 20-1
assembler reserved word 9-5, 9-6, 20-16
assembly language 9-4, 9-5
calling conventions 20-1
accessing parameters 20-10
expression differences 20-6, 20-7
procedures and functions 20-16 to 20-17
valid symbols 20-10
comments 20-2
expressions 20-5, 20-6 to 20-16
accessing Object Pascal symbols 20-9
classes categorized 20-11
invalid symbols 20-9
member selector 20-11
operators 20-11, 20-14 to 20-16
type checking 20-13 to 20-14
function result variable 20-9
instruction sizing 20-3
labels 20-2
operands 20-5 to 20-6
register names and 20-9
scope 20-11
statements 20-1, 20-2, 20-16
multiple 20-1
user-defined symbols 20-6
Assign procedure 16-4
Assigned function 4-19
interface objects 13-14
AssignFile procedure 16-1, 16-2
strings 17-4
assignment
See also assignment statements
automation objects 4-30
class types 10-5
class-reference types 10-26
compatibility 4-33
procedural types 4-20
variants 4-22, 4-23
constant parameters 9-7
Currency type and 4-7
expressions 4-4
file variables 18-8
files 16-2
functions 9-5
to variables 4-18
interface objects 13-12 to 13-13
nil values and 4-19
open-array parameters 9-9
pointer types 4-16
procedural types 5-6, 6-12
procedures to variables 4-18, 4-19, 6-13
properties 10-20
string literals 17-2
string types and 4-7, 4-11
long strings 4-9
short strings 4-8
strings 17-5, 17-6
type identifiers and 4-1
untyped pointers 6-13
value typecasts and 6-12
variables 7-1, 7-7
variants 4-20, 4-27
needing values 4-20
type compatibility 4-22, 4-23
assignment statements 7-1 to 7-2
at-sign (@) in assembler code 20-2
attributes *See* properties
automated reserved word 2-3, 10-5, 10-6
automation 13-14 to 13-16
See also OLE applications
accessing object properties 4-30
dispatch ID 10-7
method calls 4-29, 4-30, 13-14
dispatch interfaces 13-14
dual interfaces 13-16
failing 4-31

positional vs. named parameters 4-30
references 4-25, 4-29
type information 10-6
variants and 4-20, 4-21, 4-25
conversions 4-23, 4-24
wide strings and 4-11

B

$B compiler directive 6-6
base classes
interface objects 13-6, 13-8
method resolution 13-9
properties 10-25
virtual methods and 10-9
base registers 20-9
base type 4-16
begin reserved word 7-3
binary data
sending/receiving 4-28
binary notation 20-7
binary operators 6-4
defined 6-1
precision 4-4
bit masks (variants) 4-22
bitwise negation 4-26
bitwise operators 6-5
built-in assembler 20-16
blanks 2-1
defined 2-1
BlockRead procedure 16-1, 16-4
blocks 8-1 to 8-2
memory 18-1
nesting 9-1
scope 8-3
BlockWrite procedure 16-1, 16-4
Boolean expressions 4-5, 6-6
variants and 4-20, 4-23, 4-24
Boolean flag parameter 19-4
Boolean operators 6-6
Boolean type 4-5
internal formats 18-2
ordinal types and 4-3
structured statements 7-4, 7-6
variant expressions 4-25, 4-26
Boolean types 4-5 to 4-6
storage 18-2
Borland Assist program 1-2
Borland C/C++ 9-3, 12-5
brackets ([ ]) in expressions 6-11
built-in assembler 20-9
Break procedure 7-5, 11-8
BSTRs *See* wide strings
buffers 17-5, 18-8
text files 16-5
built-in assembler *See* assembler
Byte type 4-3
internal formats 18-2
ByteBool type 4-3, 4-5
internal formats 18-2
bytes 17-6, 19-2, 20-4
Boolean types and 4-5
character types and 4-4
operands and 4-4
real types and 4-7
set types and 10-6
string types and 4-8, 4-9
type boundaries 18-7
variants and 4-22

C

C/C++ calling conventions 9-3, 12-5
calling conventions 19-1 to 19-5
assembly language 20-1
accessing parameters 20-10
expression differences 20-6, 20-7
procedures and functions 20-16 to 20-17
valid symbols 20-10
automation methods 4-30, 13-14
dispatch interfaces 13-14
dual interfaces 13-16
error conditions 4-31
positional vs. named parameters 4-30
variants and 4-29
C/C++ language 12-5
Destroy method 10-15
functions 4-20, 4-31, 6-10 to 6-11, 9-5
interface objects 13-5, 19-1
message handlers 10-17
methods 10-10, 19-4
null-terminated strings 17-4
procedural types 4-18, 4-20
procedures 4-31, 7-2, 9-2 to 9-3
address-of operator and 6-13
routines in DLLs 9-3
type identity and 4-31
units 12-2, 12-4
Cardinal type 4-3
carriage-return characters 16-3
case reserved word 7-4
case sensitivity
identifiers 2-3
reserved words 2-2
case statements 7-4 to 7-5
-cc compiler directive 16-3
cdecl reserved word 9-2, 19-1
Boolean flag parameter and 19-4
Self parameter and 19-4
Char type 4-4
internal formats 18-2
long strings and 4-9
open arrays 9-10
ordinal types and 4-3
short strings and 4-8
Text type vs. 16-3
character arrays 5-8
empty strings and 9-11
initializing 5-9, 17-3
wide characters and 17-7
zero-based 4-13, 17-1
character buffers 17-5
character pointers 4-16, 17-2
comparing 6-9
indexing into 17-3
offset evaluation 6-7
operators 6-7
string literals and 17-2
character sets 4-30, 17-6
character strings 2-5 to 2-6, 17-1
See also strings
compatibility 17-2, 17-6
character types 4-4
storage 18-2
characters 2-1, 4-4
arrays of 4-13, 5-9, 17-1
as symbols 2-2
extended ASCII 2-5
identifiers and 2-3
returning values 4-4
specifying 5-8
undefined 4-8
ChDir procedure 16-1
Chr function 3-2, 4-4
circular references 12-5
packages 15-3
class methods 10-3, 10-28
class operators 10-15 to 10-16
class reserved word 10-1, 10-4, 10-28
exceptions 11-2
class types 10-1
See also class-reference types
as parameters 19-2

comparing 6-9
compatibility 10-5
declarations 10-1, 10-3, 10-4
field definitions 10-3
heritage 10-4
methods 10-3, 10-8
properties 10-4, 10-18, 10-22
visibility specifiers 10-5 to 10-7
inheritance 10-4
instantiating 10-2, 10-5
interface objects and 13-12
predefined 10-28
referencing/dereferencing 5-5, 10-3, 10-5
return values 19-3
scope 8-3, 10-4, 10-5
storage 18-9
unknown 10-6, 10-26
values 10-3
classes
ancestor vs. descendant 10-4
components defined 10-3
deriving 10-7, 10-26
default properties and 10-23
exception handling 11-5
inherited properties and 10-25
exception objects 11-1, 11-8
interface 13-6, 13-7
method compatibility 13-7
method resolution 13-9
message handling 10-17
pointers 18-9
references 10-25
class methods and 10-28
constructors and 10-12, 10-13
virtual method tables 18-9, 18-10
class-reference types 10-25 to 10-27
as parameters 19-2
constructors and 10-27
predefined 10-28
return values 19-3
storage 18-10
ClassType method 10-27
cleanup operations 11-6
Clear procedure 9-10
Close function 16-4, 16-6
CloseFile procedure 16-1, 16-2
device drivers 16-5
closing files 11-7, 16-2, 16-6
CmdLine variable 14-8
code
assembler optimizations 20-16
exporting 14-1, 14-5, 14-6, 14-9
importing 14-2 to 14-5
by name 14-2
by ordinals 14-2
testing execution 14-8
code segments 20-4
COM methods 9-8
COM objects 13-1
inheritance 13-3
comments 2-1, 2-6
built-in assembler 20-2
Comp type 4-7
internal formats 18-4
simple types and 4-2
comparisons 6-8 to 6-9
procedural values 6-13
strings 17-5
unassigned variants 4-25, 4-26
compiler
alignment 18-7
character pointers and 17-4
comments and 2-6
default properties and 10-23
DLLs and 14-4
locating identifiers 12-2
modules and 12-5
open arrays and 9-10
packages and 15-3 to 15-5
program names and 12-1
recompiling programs 12-5
untyped parameters and 9-9
virtual method tables and 18-9
compiler directives 2-6
packages 15-4, 15-5
compile-time errors 12-3
circular references 12-5
interface objects 13-13
complete evaluation 6-6
component designators 5-5
Component Object Model *See* COM
component types 4-12
array-type constants 5-8
components 10-3
accessing 10-3, 10-4, 10-5
inheritance and 10-4
installing 15-1
record types and 4-13
references 5-5
visibility specifiers 10-5 to 10-7
compound statements 7-3
for statements and 7-7
in blocks 8-2
while statements and 7-6
compressing data storage 4-12
concatenation 6-6, 17-5, 17-6
conditional jump instructions 20-3
conditional statements 7-3
console applications 16-3
const reserved word 9-7
array types 4-27, 5-8
pointer types 5-10
procedural types 5-11
record types 5-9
set types 5-10
simple types 5-7
string types 5-8
untyped parameters and 9-8
constant expressions 3-1
addresses 5-7, 5-10
constant parameters 9-7, 19-2
constants 2-3, 2-4, 3-1 to 3-2
arithmetic operations and 4-4
arrays 5-8
Boolean types 4-5
built-in assembler 20-6, 20-7 to 20-8
case statements 7-4
character size 4-4
character types 4-4
control characters and 2-5
decimal numbers and 2-5
declarations 3-1, 3-2
in blocks 8-2
in DLLs 14-3
typed 5-7 to 5-11
enumerated types 4-5
exporting 14-1
external directives 14-2
hexadecimal numbers and 2-5
interface objects 13-3, 13-4
pointer types 4-16, 5-10
private 12-3
procedural types 5-11
public 12-3
record types 4-14, 5-9

set types 5-10
setting values 5-7
structured types 5-8 to 5-10
subrange types 4-6
typed 5-7 to 5-11
unsigned 6-2
variants 4-21, 18-10
constructor reserved word 10-11, 10-12
constructors 10-2, 10-12 to 10-14
calling 19-4
class-reference types and 10-27
declaring 10-11, 10-12
defined 10-3
exception classes 11-8
exception handling 10-13, 10-14
virtual 10-27
constructs
constant expressions and 3-1
ignored by compiler 2-6
structured statements and 7-3
contains reserved word 15-2, 15-3
Continue procedure 7-5, 11-8
control characters (ASCII) 2-1
embedding in strings 2-5
control variables 7-7
controls 10-20
message processing 10-17, 10-18
conversions
ANSI to Unicode 4-24
binary operators 4-4
exception handling 11-3
ordinal types 6-12
string types 4-7, 4-11
strings 11-3, 17-5, 17-7
Unicode to ANSI 4-24
variants 4-22 to 4-25
copying strings 4-9, 17-5
CPU registers 19-1, 19-3
assembler and 20-2, 20-8, 20-11, 20-12
preserving 19-5, 20-2
CreateOleObject function 4-29
creating objects 10-3, 10-12
class-reference types and 10-27
Currency type 4-7
internal formats 18-4
pointers 4-17
variant expressions 4-25, 4-26
CurrToStr function 9-12
customer assistance 1-2

D

data 10-3
exception handling 12-4
exporting 14-1
internal formats 18-2 to 18-11
missing 4-20
storing 18-8
transfers 16-4
unknown 4-20
data segments 20-2, 20-4
data types *See* types
Date variant type 4-25, 4-26
dates 4-20, 4-23
DB directive 20-4
DBCS (double-byte character set) 17-6
.DCP files 15-4
.DCU files 15-4
DD directive 20-4
deallocating memory 18-1
objects and 10-3
debugging 18-2
decimal notation 2-4
decimals 2-5
Currency type and 4-7
declarations 8-1
See also syntax diagrams
arrays 4-13
class methods 10-28
class types 10-1, 10-3, 10-4
field definitions 10-3
heritage 10-4
methods 10-3, 10-8
properties 10-4, 10-18, 10-22
visibility specifiers 10-5 to 10-7
class-reference types 10-26
constants 3-1, 3-2, 4-6
in blocks 8-2
in DLLs 14-3
typed 5-7 to 5-11
constructors/destructors 10-11, 10-12, 10-14
exceptions 11-2
forward 9-3
class types 10-3, 10-4
interface objects 13-5
methods 10-10
functions 6-11, 9-5 to 9-6
built-in assembler 20-16 to 20-17
in blocks 8-2
in DLLs 14-3
interface objects 13-1, 13-2 to 13-3, 13-5
classes 13-6
identifiers 13-4
labels 8-1
message handlers 10-16
methods 10-10, 10-11
class types and 10-3
nested procedures and functions 4-20
parameters 9-6
open-array 9-9, 9-11
procedures 7-2, 9-1 to 9-5
built-in assembler 20-16 to 20-17
in blocks 8-2
in DLLs 14-3
types 4-1, 4-12, 4-17, 4-34
automation objects 4-29
in blocks 8-2
in DLLs 14-3
long strings 4-9, 4-10
pointers 4-17
procedural 4-18
record types 4-13
short strings 4-8
units 12-2, 12-3
finalization 12-4
implementation 12-3
initialization 12-3
variables 4-1, 5-1, 8-2
assembler and 20-4, 20-7, 20-11
assembly member selector 20-11
component designators 5-5
field designators 5-4
pointer types and 4-17
pointers as 5-5
qualified names and 5-3
default properties 10-22, 10-24
interfaces 13-5
default reserved word 10-22, 10-24
interface objects and 13-5
property overrides and 10-25

DefaultHandler method 10-17, 10-18
DELPHIMM.DLL 14-9
denormals 18-3
$DENYPACKAGEUNIT compiler directive 15-4
dereferencing
  class types 10-3
  pointer types 4-16, 5-3
  structured types 5-3
derived classes 10-7, 10-26
  default properties and 10-23
  exception handling 11-5
  inherited properties and 10-25
descendants 8-3, 10-4, 10-5
  abstract methods and 10-10
  class-reference types and 10-26
  inherited fields 18-9
  interface objects 13-3
  virtual methods and 10-8, 10-9
$DESIGNONLY compiler directive 15-4
design-time packages 15-1
  generating 15-4
Destroy method 10-13, 10-14
  exceptions 11-2
  Free method vs. 10-15
destroying exceptions 11-2, 11-8
destroying objects 10-3, 10-14
destructor reserved word 10-11, 10-12
destructors 10-2, 10-14 to 10-15
  calling 19-4
  checking for nil values 10-15
  declaring 10-11, 10-14
  defined 10-3
device drivers 16-4 to 16-6
devices 16-2
  input/output 16-3
difference 6-7
digits 2-4
  defined 2-1
  floating-point numbers 4-6
  identifiers and 2-3
  labels and 2-5
direction flag 19-5
directives 2-3, 9-2
  assembler, defined 20-4
dispatch ID (automation) 10-7
dispatch interfaces 13-14
Dispatch method 10-17
dispatched messages 10-16, 10-17
dispid reserved word 10-7
Dispose procedure 14-9
div operator 6-4
  variant expressions 4-25
divide-by-zero errors 11-4
division 6-4
DLL_PROCESS_DETACH constant 14-9
DLL_THREAD_ATTACH constant 14-9
DLL_THREAD_DETACH constant 14-9
DLLProc variable 14-9
DLLs 14-1
  accessing procedures and functions 14-2 to 14-5
  calling routines 9-3
  compiling 14-4
  creating 14-5 to 14-8
  defined 14-1
  entry point 14-9
  error handling 14-7, 14-8
  exiting 14-7
  getting entries 14-7
  importing procedures and functions 9-4
  initializing 14-7
  loading 14-7
  memory management 14-8, 14-9
  modifying 14-4
  multiple 14-3, 14-8
  packages and 15-1
  variables and 14-3, 14-8
documentation 1-1
  printing conventions 1-2
dollar sign ($) prefix 2-4, 2-6
domain 10-4
Double type 4-7
  internal formats 18-4
  simple types and 4-2
  variant expressions 4-25, 4-26
double words
  Boolean types and 4-5
  set types and 10-6
double-byte character sets 17-6
doublewords 19-2, 20-4
downto reserved word 7-7
.DPK files 15-1
  generating 15-3
.DPL files 15-1, 15-4
drivers 16-4
dual interfaces 13-16
duplicate names
  interface objects 13-8
  procedures and functions 12-3
  unit calls 8-4, 12-2
DW directive 20-4
dynamic imports 14-4
dynamic linking 14-1
dynamic methods 10-9
dynamic reserved word 10-9
dynamic variables
  See also variables
  defined 5-7
  local vs. 5-7
dynamic-link libraries *See* DLLs

E

editors 15-1
EInOutError class 11-2
else reserved word 7-4
  exception handlers 11-4, 11-5
EMathError class 11-5, 11-7
embedding control characters in strings 2-5
empty sets 4-15, 6-11
empty strings 4-9, 9-11
end reserved word 7-3
end-of-line characters 2-1
end-of-line markers 16-3
engineering notation 2-4
entry code (assembler) 20-16
entry point (DLLs) 14-9
enumerated types 4-5
  storage 18-3
  subrange types and 4-6
Eof function 16-1
  device drivers 16-5
EOleError exception 4-31
Eoln function 16-1
  device drivers 16-5
EQU directives 20-1
equal precedence 6-1
equality 6-8
ERangeError class 11-3
Erase procedure 16-1
error codes 4-22, 11-2, 19-6
error messages 19-6
ErrorAddr variable 19-6
errors 11-1
  automation methods 4-30, 4-31, 13-16

circular references 12-5
exceptions failing 11-4
insufficient memory 5-2
interface objects 13-13
nil functions and procedures 4-19
pointer types and 4-16
program termination 19-6
type-mismatch 10-15, 10-16
unit implementation 12-3
variant conversions 4-23, 4-24
EStackOverflow exception 5-2
EVariantError exception 4-23, 4-27
OLE automation 4-25
examples
abstract methods 10-10
array properties 10-21, 10-22
array types 4-12
as constants 5-8, 5-9
assembler directives 20-4, 20-5
assembly language
entry/exit code 20-16
function declarations 20-16
assembly statements 20-2, 20-3, 20-5
expressions 20-6, 20-7, 20-10, 20-11, 20-12
function result variable 20-9
parameters 20-10
type checking 20-13
automation objects 4-29, 4-30
character strings 2-5
class methods 10-28
class types 10-5
comments 2-6
constants 3-1, 3-2
constructors 10-13
destructors 10-14
DLLs 14-5, 14-6, 14-7
exceptions 11-2, 11-6
raise statements 11-3
try...except statements 11-4, 11-5
try...finally statements 11-7
exit procedures 19-5
expressions 6-3
factors 6-2
set constructors 6-12
terms 6-3
value typecasts 6-12
external declarations 9-4
forward declarations 9-3, 10-5, 10-11
function calls 6-11
function declarations 9-6
global procedures 4-18, 4-19
identifiers 2-3
import units 14-3, 14-4
importing procedures and functions 14-2, 14-4
interface objects 13-5, 13-8
assignment 13-12, 13-13
declarations 13-3, 13-6
method compatibility 13-7
method resolution 13-8, 13-10
unique GUIDs 13-4
message handlers 10-17
method pointers 4-18, 4-19
null-terminated strings 4-10, 17-1, 17-3, 17-4
string literals and 17-2
packages 15-2
pointer-type constants 5-10
polymorphism 10-27
procedural types 6-12, 6-13
as contants 5-11
procedures 9-2
property overrides 10-25
record types 4-14
as constants 5-9
resource strings 3-2
set-type constants 5-10
simple statements 7-2
static methods 10-8
structured statements 7-3, 7-4, 7-5, 7-6, 7-8, 7-9
subrange types 4-6
type declarations 4-34
untyped parameters 9-8
variable declarations 5-1, 5-5
field designators 5-4
pointer variables 5-5
procedural types and 5-6
qualified names 5-3
variant types 4-21
arrays 4-27, 4-28
virtual methods 10-8, 10-9
except reserved word 11-3
Exception class 11-2, 11-8
exception handlers *See* exceptions
exception statements 7-10
exceptions 11-1
cleanup operations 11-6
constructors 10-13, 10-14
declarations 11-2
destroying 11-2, 11-8
DLLs 14-7, 14-8
files 16-2, 16-3
interface objects 13-12
nesting 11-6
OLE applications 13-16
predefined 11-8
raising 11-2, 11-4, 11-8
re-raising 11-6, 11-7
scope 11-5
unit calls and 12-4
executable files 19-5
executable modules 14-1
executing programs 7-2
Exit procedure 11-8
exit procedures 14-7, 19-5 to 19-6
assembly language 20-16
ExitCode variable 14-7, 19-6
ExitProc variable 14-7, 14-8, 19-5
exponents 2-4, 2-5, 18-3
exporting source code 14-1, 14-5, 14-6, 14-9
exports directive 14-5, 14-6
exports reserved word 8-2, 14-6
expressions 6-1, 9-7, 9-11
arithmetic 6-4 to 6-5
arrays and 5-4
assembly language 20-5, 20-6 to 20-16
accessing Object Pascal symbols 20-9
classes categorized 20-11
invalid symbols 20-9
member selector 20-11
operators 20-11, 20-14 to 20-16
type checking 20-13 to 20-14
assignment statements 4-4
automation objects and 4-30
Boolean types and 4-6
constants and 3-1, 5-7, 5-10
Currency type and 4-7
exceptions 11-2
interface objects 13-14
logical 6-5 to 6-6
pointer types 5-3
procedural types and 6-12 to 6-14

property references and 10-20, 10-21
relational 6-8 to 6-9
Result variable and 9-6
set constructors 6-11
set operators 6-7
simple 6-3
string 6-6, 17-5
string types and 4-7, 4-9, 4-11
structured statements 7-4, 7-6
subrange types and 4-6
value typecasts 6-12
variant records 9-13
variants and 4-22, 4-25 to 4-26
operands 6-8
extended ASCII characters 2-5
extended syntax
string compatibility 2-6
variables references and 5-3
Extended type 4-7
arithmetic expressions 6-5
internal formats 18-4
pointers 4-17
simple types and 4-2
extended values 6-12
external directives 9-4, 14-2, 14-4
external files 16-2
external packages 15-2
duplicate references 15-3
external procedures and functions 20-1
external reserved word 9-4, 9-6
DLLs and 14-2

F

factors 6-2
addresses 6-10
multiplication 6-3
field designators 5-3, 5-4
field names 4-13
fields 2-3, 4-14
access specifiers and 10-21
accessing 5-3
class types and 10-3, 10-6
index specifiers and 10-23
interface objects and 13-3
nil values 10-14
objects and 10-18, 18-9
pointers 9-12
record types and 4-13, 9-12
referencing 7-9
setting values 5-9
storing data 18-8
file buffers 16-5, 18-8
file handles 18-8
file modes 16-5, 16-6, 18-8
file names 18-8
file reserved word 16-4
file types 4-15, 9-8, 16-2
constants as 5-8
function calls and 9-6
property definitions and 10-19
storage 18-7
file variables 16-2, 16-3
assignment 18-8
FilePos function 16-1, 16-2
files
assignment 16-2
closing 11-7, 16-2, 16-6
exception routines 11-7
formatted text 16-3
functions for 16-1, 16-4
loading 4-28
low-level access 4-15, 16-4
naming 18-8
opening 11-7, 16-3, 16-5
position 16-2
procedures for 16-1
size 16-2
untyped 4-15, 18-7
FileSize function 16-1, 16-2
finalization reserved word 12-4, 14-8
finally reserved word 11-7
FindFirst function 11-6
FindNext function 11-6
floating-point numbers 4-6
floating-point registers 20-9
floating-point types
storage 18-3 to 18-4
FloatToStr function 9-12
Flush procedure 16-1, 16-4, 16-5
fmClosed constant 18-8
fmInOut constant 16-5, 18-8
fmInput constant 16-5, 18-8
fmOutput constant 16-5, 16-6, 18-8
for reserved word 7-5, 7-7
for statements 7-7 to 7-8
formal parameters 7-2, 9-6
types listed 9-7
formatted text files 16-3
formatting strings 17-5
forward declarations 9-3
class types 10-3, 10-4
interface objects 13-5
methods 10-10
forward references 10-4
forward reserved word 9-3, 9-6
Free method 10-15
freeing memory *See* deallocating memory
FreeMem procedure 14-9
function reserved word 4-18, 9-5
functions 2-3, 9-1
See also procedures
addresses 6-10
arrays and 9-9
assigning to variables 4-18
Boolean expressions 6-6
calling 4-20, 6-10 to 6-11, 9-5
type identity and 4-31
cleanup operations 11-6
constant expressions and 3-2
declarations 6-11, 9-5 to 9-6
built-in assembler 20-16 to 20-17
forward directives and 9-3
in blocks 8-2
in DLLs 14-3
DLLs and 14-1, 14-3
exporting 14-1, 14-5, 14-6, 14-9
external 20-1
importing 9-4, 14-2 to 14-5
interface objects and 13-3, 13-5
method pointers and 4-20
naming 5-11, 9-5, 12-3
nested 4-20
null-terminated strings and 4-10
OLE applications and 4-11
pointers 18-8
public 12-3
qualified method identifiers 10-11
recursive 9-3, 9-5
references 4-18, 12-3
return values 7-1, 9-5, 9-6
built-in assembler 20-17
undefined 9-5
returning values 19-3
typed constants and 5-7

variables local to 5-2
fundamental types 4-1
characters 4-4
integers 4-3
pointers 4-16

G

$G compiler directive 15-5
general-purpose registers 20-8
generic types 4-1
characters 4-4
integers 4-3
pointers 4-16
GetDir procedure 16-1
GetHeapStatus function 18-2
GetMem procedure 4-16, 5-5, 14-9
GetMemoryManager procedure 18-2
GetProcAddress function 6-13
getters 13-3
global procedures 4-18
method pointers and 4-20
global variables 5-1
DLLs and 14-8
typed constants and 5-7
globally unique identifiers 13-4
goto reserved word 7-2
goto statements 7-2
labels and 2-5, 7-1
greater-than operator 6-8
guarded typecasts 10-15
GUIDs 13-4

H

$H compiler directive 4-8
long strings 4-9
Halt procedure 19-5, 19-6
handles 18-8
DLLs 14-8
heap-based variables 5-7
HelpContext property 11-8
hex digits, defined 2-1
hexadecimal notation 2-1, 2-4, 20-7
hexadecimal numbers 2-4
as constants 2-5
Hi function 3-2
High function 4-2
arrays and 4-13
constants and 3-2
open arrays 9-10
string types and 4-8
high-order bytes 20-15
HInstance variable 14-8
host type 4-6
hosts 12-3

I

$I compiler directive 16-3
I/O 9-8, 16-2 to 16-4
devices 16-4
error codes 11-2
error-checking 16-2 to 16-3
low-level file access 4-15, 16-4
pointers 18-8
standard procedures 16-1
identifiers 2-3
arrays and 4-13
as labels 2-5
assembly language 20-11
and user-defined 20-6
automation objects 4-30
constants and 3-1
directives and 2-3
DLLs and 14-6, 14-7
duplicating 12-3
interface objects 13-8
unit calls 8-4, 12-2
exceptions 11-5
functions 5-11, 9-5, 12-3
inherited directives and 10-11
interface objects 13-4
methods 10-11
multiple instances 2-3
overriding 20-15
packages 15-2
procedures 5-11, 9-2, 12-3
programs 12-1
scope 8-3 to 8-4
separators and 2-1
type declarations 4-1, 4-2, 4-5
units 12-2
variables and 5-1, 5-3
identity 4-31
IDispatch interface
dispatch declarations 13-14
dual interfaces 13-16
implementing 13-14 to 13-16
type compatibility 13-12
typecasting 13-12
IDispatch interface pointer 4-22
if reserved word 7-4
if statements 7-4
is operator and 10-15
immediate values (assembler) 20-12
$IMPLICITBUILD compiler directive 15-4
import units 14-3
IMPORTEDDATA compiler directive 15-4
importing procedures and functions 9-4
importing source code 14-2 to 14-5
by name 14-2
by ordinals 14-2
importing SysUtils unit 11-3
in operator 6-8, 6-9
index register 20-9
index reserved word 10-23
DLLs and 14-6
external directives and 14-2
index specifiers 10-23
indexed properties 10-21
indexes 4-12
array properties and 10-22
assembly registers 20-9
character pointers 17-3
DLLs and 14-7
field designators and 5-3
multidimensional arrays 5-4
open-array parameters 9-9
string arrays 5-4
variant types and 4-27
indirect references 12-4
inequality 6-8
infinities 18-3
inheritance 10-4
COM objects 13-3
exceptions 11-2
interface objects 13-3 to 13-4, 13-7
inherited classes
interface objects 13-9
inherited properties 10-25
inherited reserved word 10-4
Boolean flag parameter and 19-4
constructors and 10-12
message handlers 10-17
method calls 10-11
initialization reserved word 12-3, 14-8
initializing
arrays 5-9, 17-3
DLLs 14-7
interface objects 13-14
long strings 4-9
objects 10-12
pointer types 4-17

units 12-2, 12-3
variants 4-21
inline procedures and functions 12-3
inline reserved word 12-3
InOut function 16-4, 16-5
input files 16-3
input *See* I/O
Input variable 16-3
installing components 15-1
instances 10-2, 10-5
method calls and 10-10
instruction opcodes 20-3
instructions 20-12
Integer type 4-3
variant expressions 4-25, 4-26
integer types 4-3 to 4-4
arithmetic expressions 6-5
logical expressions 6-5
notating 2-4
ordinality and 4-2
storage 18-2
integers 4-3
as constants 2-5
conversions 11-3
typecasting 4-4
variants and 4-20, 4-23
interface objects 13-1
assignment 13-12 to 13-13
automation 13-14 to 13-16
calling 13-5, 19-1
declarations 13-1, 13-2 to 13-3
classes 13-6
forward references 13-5
identifiers 13-4
default properties 13-5
duplicate names 13-8
inheritance 13-3 to 13-4, 13-7
initializing 13-14
mapping methods 13-7 to 13-8
overriding 13-8
member visibility 13-5
method resolution 13-8 to 13-11
restrictions 13-9
naming 13-4
references 13-1, 13-11 to 13-14
access specifiers and 13-5
counters 13-11, 13-13
getting 13-12
querying 13-13
type compatibility 13-11
typecasting 13-12
interface reserved word 13-2, 13-5
forward declarations 9-3
interfaces 13-1
implementing 13-1, 13-6 to 13-11
COM objects and 13-4
error conditions 13-13
multiple references 13-11
mutually dependent 13-5
name conflicts 13-8
OLE applications 13-14 to 13-16
reimplementing 13-10
unique identifiers 13-4
internal data formats 18-2 to 18-11
international character sets 4-30
interrupts 11-1
intersection 6-7
IntToStr function 9-12
invalid symbols
built-in assembler 20-9
Invoke method 13-14
IOResult function 16-1, 16-3
is operator 10-15
class-reference types and 10-26
IsLibrary variable 14-8
iterations 7-5
IUnknown interface 13-3 to 13-4, 13-7
pointer 4-22
reference queries and 13-14
type compatibility 13-12
typecasting 13-12

J

$J compiler directive 5-7
Java applications 13-1
JMP instruction 20-3
jump instructions 20-3

K

keywords *See* reserved words

L

labels 7-1, 8-1
assembly language 20-2
defined 2-5
scope 8-3 to 8-4
separators and 2-1
-LE <path> compiler directive 15-5
leading zeros 2-5
left brace ({) symbol 2-2
left bracket ([) symbol 2-2
Length function 4-9
constants and 3-2
less-than operator 6-8
letters 2-1
identifiers and 2-3
$L<filename> compiler directive 9-4
libraries 12-1
See also DLLs
packages 15-1
library reserved word 14-5
linefeed characters 16-3
linked lists 4-17
linking 20-1
relocation and 20-12
literal strings 17-2
memory management 18-5
wide characters and 17-6
-LN <path> compiler directive 15-5
Lo function 3-2
loading DLLs 14-7
loading files 4-28
local labels 20-2
local variables 5-1, 5-2
built-in assembler 20-10
dynamic vs. 5-7
function calls 9-6
read-only 9-7
typed constants and 5-7
value parameters as 9-7
logical expressions 6-5 to 6-6
logical negation 4-26
logical operators
bitwise expressions 6-5
Boolean operations 6-6
code generation models 6-6
long string types 4-7, 4-8, 17-5
arrays and 5-4
as parameters 19-2
constants as 5-8
records and 4-14, 9-12
storage 18-5
LongBool type 4-3, 4-5
internal formats 18-2
Longint type 4-3
internal formats 18-2
lookup tables 10-22
Low function 4-2

arrays and 4-13
constants and 3-2
open arrays 9-10
string types and 4-8
lowercase conversions 17-5
lowercase letters 2-2
low-order bytes 20-15
lsb (defined) 18-3
-LU <package> compiler directive 15-5

M

$M compiler directive 10-6
machine code 20-11
Macro Assembler 20-1
makes 12-5
MakeStr function 9-12
math exceptions 11-4, 11-7
$MAXSTACKSIZE compiler directive 5-2
member of operator 6-8, 6-9
members 10-3
interface objects and 13-5
memory 18-1
allocating 5-2, 18-1
dynamic variables 4-16
objects and 10-3
array access 4-17
DLLs 14-8, 14-9
error conditions 5-2
locations 20-12
long strings and 4-10
procedural variables 6-14
reference counting and 4-9, 4-11
references 20-12, 20-15
variants and 4-21
memory manager 18-1 to 18-2
memory mapped files 14-8
message dispatching 10-17
message handlers 10-16 to 10-18
declarations 10-16
message IDs 10-16
Message property 11-8
message reserved word 10-16
messages 11-8
Messages unit 10-17
metaclasses 10-26
method designators 5-5, 10-10
method pointers 4-18
global procedures and 4-20
return values 19-3
methods 10-2, 10-3
abstract 10-10
access specifiers and 10-21, 10-22
addresses 6-10
automated directives and 10-6
automation objects 4-29, 4-30, 13-14
calls failing 4-31
dispatch interfaces 13-14
dual interfaces 13-16
positional vs. named parameters 4-30
calling 10-10, 19-4
class 10-28
component references and 5-5, 10-4
declarations 10-10, 10-11
class types and 10-3
forward 10-10
dynamic 10-9
interface objects and 13-1, 13-3, 13-5
mapping 13-7 to 13-8
overriding mapping 13-8
message directive and 10-16
naming 10-11
out parameters and 9-8
overriding 10-8, 10-10
message handlers 10-17
pointers 18-8
referencing 10-10, 13-11
scope 10-11
static 10-8
virtual 10-8 to 10-9
constructors as 10-14
$MINSTACKSIZE compiler directive 5-2
missing data 4-20
MkDir procedure 16-1
mod operator 6-4
variant expressions 4-25
modular programming 12-1, 12-4
circular references and 12-5
modules
compiling 12-5
DLLs as 14-1
importing procedures and functions 14-2 to 14-5
interface objects and 13-3, 13-5
msb (defined) 18-3
multidimensional arrays 4-12
array-type constants and 5-9
indexing 5-4
variants and 4-27, 4-28
multi-language programming 14-1
interface objects and 13-5
multiple identifiers 2-3
multiplication 6-3, 6-4
precedence 6-1
multi-threaded applications 14-9

N

name reserved word
DLLs and 14-7
external directives and 14-2
named parameters 4-30
names 2-3
See also identifiers
duplicate
interface objects 13-8
procedures and functions 12-3
unit calls 8-4, 12-2
file 18-8
importing by 14-2
qualified 2-3, 5-3
uses clause and 8-4
NaNs 18-3
near returns 20-3
negation 6-6
nested blocks 9-1
nested exceptions 11-6
nested functions 4-20
nested procedures 4-20
New procedure 4-16, 5-5, 14-9
nil reserved word 4-16
exit procedures 19-6
function calls 4-19
procedures 4-19
variables and 5-5
nil values
checking for 10-15
class types and 10-2, 10-27
DLLs 14-8
interface objects 13-11, 13-14
object fields 10-14
pointers 5-5
procedural types 5-11
procedures 4-19, 4-20
testing for 4-19
nodefault reserved word 10-24
property overrides and 10-25
nonzero values
Boolean types and 4-6
not operator

bitwise operations 6-5
Boolean operations 6-6
variant expressions 4-25, 4-26
null characters 4-10, 17-1
arrays 9-11
long strings 18-5
wide strings 18-6
Null constant 4-20
variant expressions 4-25, 4-26
null strings 2-5
null values 4-20
arrays and 5-9
variants 4-25, 4-26
null-terminated strings 4-10, 17-1 to 17-7
defined 17-1
OLE and 4-11
pointers 4-17
storing 4-13
number sign (#) in code 2-5
numbers 2-4 to 2-5, 4-3, 18-3
See also integers; ordinal numbers; real numbers
floating-point 4-6
separators and 2-1
variants and 4-20, 4-23
numeric constants 3-1
built-in assembler 20-7
precision 4-4
numeric types 4-2, 4-3, 4-6

O

object files 9-4
object interfaces *See* interface objects; interfaces
Object Pascal
assembly calls 20-1
accessing parameters 20-10
expression differences 20-6, 20-7
procedures and functions 20-16 to 20-17
valid symbols 20-10
case sensitivity 2-2, 2-3
circular references and 12-5
internal data formats 18-2
memory management features 18-1
modular programming 12-1, 12-4
naming conventions 2-3
null-terminated strings and 17-1
redeclaring predefined identifiers 8-4
object types
directives and 2-3
file types and 4-15
objects 10-3
accessing 13-1, 13-13
arrays 5-3
class-reference types and 10-27
component designators 5-5
creating/destroying 10-3, 10-12
defined 10-2
exceptions and 11-1, 11-8
exporting 14-1
initializing 10-12
interface *See* interface objects
internal formats 18-9
references 4-18, 7-9, 10-12
class methods and 10-28
class types and 10-2
Self identifier and 10-3
scope 8-3
setting attributes 10-4, 10-18
unknown types 10-26, 10-27
octal notation 20-7
Odd function 3-2
offsets 5-7
difference 6-7
incrementing/decrementing 6-7
record types 18-7
virtual method tables 18-9
wide characters 17-7
OLE applications
automated properties 4-25, 4-29
accessing 4-30
string types 4-11
automation dispatch ID 10-7
automation interfaces 13-14 to 13-16
creating servers 10-6
exception handling 13-16
method calls 4-29, 4-30, 13-14
dispatch interfaces 13-14
dual interfaces 13-16
failing 4-31
positional vs. named parameters 4-30
sending/receiving binary data 4-28
string types 4-8, 4-11
variants and 4-20, 4-21, 4-25
conversions 4-23, 4-24
OLE Object Model 18-9
OleAuto unit 4-29
OleVariant type 4-26
on...do exception handlers 11-4, 11-5
opcodes 20-1, 20-2
assembler supported 20-3
open arrays 9-9, 9-11
interface objects 13-14
Open function 16-4, 16-5
open-array parameters 4-13, 9-9 to 9-13
constructor 9-11, 9-13
passing 19-2
variant type 9-11
opening files 11-7, 16-3, 16-5
operands 6-1
addresses 6-10
assembler directives 20-4
assembly language 20-5 to 20-6
register names and 20-9
binary operators 6-4
Boolean expressions 6-6
character pointers 6-7
function calls 9-5
logical operators 6-5, 6-6
ordinal types and 6-4
set operators 6-7
string operators 6-6
unary operators 6-4
variants 6-8
operators 6-1, 6-4 to 6-10
See also operands
built-in assembler 20-11, 20-14 to 20-16
class references 10-15
code generation models 6-6
object references 10-15
precedence 6-1, 10-15, 10-16
precision 4-4
types 6-4
or operator
bitwise operations 6-5
Boolean operations 6-6
variant expressions 4-25
Ord function 4-2
Boolean types and 4-5
character types and 4-4
constants and 3-2

enumerated types and 4-5
string types and 4-8
ordered sets 4-2, 4-5
ordinal numbers 4-2
Boolean types and 4-6
ordinal types 4-2 to 4-6
arithmetic expressions 6-4
arrays and 4-12
as parameters 19-2
conversions 6-12
properties and 10-24
record types and 4-14
return values 19-3
user-defined 4-5, 4-6
value typecasts 6-12
ordinality 4-2
characters and 4-4
enumerated types 4-5
set types 4-15
true/false values 4-5
out parameters 9-8
out reserved word 9-7
untyped parameters and 9-8
output files 16-3
output *See* I/O
Output variable 16-3
overflow exceptions 11-4
override reserve word 10-8
overriding identifiers 20-15
overriding methods 10-8, 10-10
message handlers 10-17
overriding property definitions 10-25

P

package reserved word 15-2
packages 15-1
compiling 15-3 to 15-5
exceptions and 11-3
naming 15-2
references 15-2, 15-3
saving 15-1
packed arrays 2-6
packed records 18-7
packed reserved word 4-12, 18-7
packed string types 4-13
comparing 6-9
constants as 5-8
PAnsiChar type 4-17
arrays and 5-4
PAnsiString type 4-17
parameters 7-2, 9-6 to 9-13, 19-1, 19-2
array properties 10-22
automation objects 4-30
built-in assembler 20-10
constant vs. value 9-7
declaring 9-6
exceptions 11-2
exit procedures 19-5
function calls 6-10, 9-5
functions as 4-18
I/O routines 16-3, 16-4
index specifiers and 10-23
interface GUIDs 13-4, 13-5
null-terminated strings and 4-10
open-array 4-13, 9-9 to 9-13
out vs. variable 9-8
passing 9-2, 9-7
by reference 4-30, 19-2
by value 4-30, 4-33, 19-2
in reverse order 19-3
procedural type compatibility 4-20, 4-31
procedures as 4-18, 6-13
programs 12-1
properties as 10-20
read-only 9-8
removing from stack 9-2
string literals and 17-2
types listed 9-7
untyped 9-8 to 9-9
open arrays and 9-9
partial evaluation 6-6
pascal reserved word 9-2, 9-3, 19-1
Boolean flag parameter and 19-4
Self parameter and 19-4
passing by reference 4-30, 19-2
passing by value 4-30, 4-33, 19-2
passing return values 9-8
PByteArray type 4-17
PChar type 17-2
arrays and 4-13, 5-4
character strings and 2-6
compatibility 17-2
long strings and 4-10
typed constants and 5-10
PCurrency type 4-17
PExtended type 4-17
pointer (^) symbol 5-5
error conditions 4-16
field designators and 5-3
object references 10-3
Pointer function 4-10
Pointer type 4-16
internal formats 18-5
pointer types 4-16 to 4-17
as parameters 19-2
constants as 5-10
expressions and 5-3
references 5-3
return values 19-3
scope 4-1
storage 18-5
pointers 4-16, 4-17
address-of operator and 6-10
assigning values 6-13
built-in assembler 20-10
class types and 10-2
classes 18-9
comparing 6-9
fields 9-12
functions 18-8
global procedures 4-18, 4-19
I/O routines 18-8
methods 4-18, 4-19, 18-8
procedural types 4-20, 6-13
procedures 18-8
setting values 5-10
stack 20-2
strings 4-9, 4-10
type-checked 4-32
undefined 5-5
untyped 4-16
setting values 6-13
variables as 5-5
virtual method tables 18-9, 18-10
polymorphism 10-9, 10-14
defined 10-27
positional parameters 4-30
pound character (#) in code 2-5
precedence 6-1
built-in assembler 20-14
class operators 10-15, 10-16
precision 4-4
real types 4-7
Pred function 4-2
constants and 3-2
predecessors 4-2
predefined exception classes 11-8
predefined identifiers 8-4
prefix opcodes 20-2
printing conventions (documentation) 1-2
private access 12-3
DLLs 14-8
Private field 18-8
private reserved word 2-3, 10-5, 10-7

PROC directives 20-1
procedural types 4-18 to 4-20
  compatibility 4-20
  constants as 5-11
  expressions and 6-12 to 6-14
  storage 18-8
  variable typecasts and 5-6
procedural values 4-19, 4-20
  comparing 6-13
  setting 6-12
procedure reserved word 4-18, 9-2
procedure statements 7-2
procedures 2-3, 9-1
  addresses 6-10
  arrays and 9-9
  assigning to variables 4-18, 4-19, 6-13
  calling 4-20, 7-2, 9-2 to 9-3
    address-of operator and 6-13
    type identity and 4-31
  cleanup operations 11-6
  declarations 7-2, 9-1 to 9-5
    built-in assembler 20-16 to 20-17
    in blocks 8-2
    in DLLs 14-3
  DLLs and 14-1
  exit 14-7, 19-5 to 19-6
    assembly language 20-16
  exporting 14-1, 14-5, 14-6, 14-9
  external 20-1
  importing 9-4, 14-2 to 14-5
  interface objects and 13-3, 13-5
  memory addresses 6-14
  naming 5-11, 9-2, 12-3
  nested 4-20
  null-terminated strings 17-4
  pointers 18-8
  public 12-3
  qualified method identifiers 10-11
  recursive 9-2, 9-3
  references 4-18, 12-3
  setting values 6-12
  typed constants and 5-7
  variables local to 5-2
product registration 1-2
programming language *See* Object Pascal
programs 2-3, 12-1 to 12-5, 19-1
  adding units 12-2
  circular references 12-5
  controlling flow 7-5
  exit procedures 14-8, 19-5
  indirect references 12-4
  large 12-1
  linking 20-1
    relocation and 20-12
  multiple DLLs and 14-3
  naming 12-1
  recompiling 12-5
  running 7-2
  static vs. dynamic imports 14-4
  static vs. dynamic linking 14-1
properties 10-18 to 10-25
  access specifiers 10-21, 10-22
  accessing multiple 10-23
  array 10-21 to 10-23
  automated directives and 10-6
  automation objects 4-25, 4-29
    accessing 4-30
  default 10-22, 10-24
    interfaces 13-5
  defined 10-4
  defining 10-18, 10-19, 10-21
  index specifiers 10-23
  interface objects 13-3, 13-5
  overriding 10-25
  published 10-6, 10-24
  read-only 10-21
  references 10-20
  saving 10-24
  storage specifiers 10-24
  write-only 10-21
property editors 15-1
property reserved word 10-25
protected reserved word 2-3, 10-5, 10-7
PShortString type 4-17
PTextBuf type 4-17
Ptr function 4-16, 5-5
public access 12-3
  interface objects 13-5
public reserved word 2-3, 10-5, 10-6
published properties 10-6, 10-24
published reserved word 2-3, 10-5, 10-6
PVariant type 4-17
PVarRec type 4-17
PWideChar type 4-17
  arrays and 5-4
PWordArray type 4-17

Q

qualified method identifiers 10-11
qualified names 2-3, 5-3
  uses clause and 8-4
queries (interface objects) 13-13
QueryInterface method 13-4, 13-7
quoted strings 2-6
quotients 6-5

R

$R compiler directive 4-2
raise reserved word 11-2
raise statements 7-10, 11-2 to 11-3
range checking 11-3
  character pointers 17-4
  ordinal types and 4-2
ranges 4-6
  case statements 7-5
  Comp type 18-4
  Currency type 18-4
  integer types 4-3
  property index specifiers 10-23
  real types 4-7
  set types 4-15
read directives 10-20, 10-21
  access specifiers and 10-21, 10-22
  index specifiers and 10-23
  interface objects 13-5
  property overrides and 10-25
Read procedure 16-1, 16-2
  device drivers 16-5
  strings 17-4
  text files 16-3
  untyped files 16-4
read reserved word 10-21, 10-23, 10-25
Readln procedure 16-1
  device drivers 16-5
  strings 17-4
read-only files 16-2

read-only parameters 9-8
read-only properties 10-21
read-only variables 5-7, 9-7
read-write properties 10-21
real numbers 4-6, 18-3
  as constants 2-5
  variants and 4-20, 4-23
Real type 4-7
  internal formats 18-3
  simple types and 4-2
real types 4-6 to 4-7
  arithmetic expressions 6-5
  as parameters 19-2
  backward compatible 4-7
  notating 2-4
  return values 19-3
recompiling packages 15-3, 15-4, 15-5
record types 4-13 to 4-15
  built-in assembler 20-11
  constants as 5-8, 5-9
  files 16-2
  long strings and 4-10
  return values 19-3
  storage 18-7
  variant 9-11
records 4-13
  arrays 5-3
  as parameters 19-2
  field designators 5-3, 5-4
  fixed fields 4-14
  linked lists 4-17
  referencing fields 7-9
  scope 8-3
  value type constants 9-12, 9-13
  variants and 5-10
recursion 9-2, 9-5
  forward declarations and 9-3
redeclaration of variables 5-1
redeclaring predefined identifiers 8-4
redefining reserved words 2-2
redefining standard directives 2-3
references 7-9
  arrays 4-13, 5-3
  built-in assembler 20-10
  circular 12-5
    packages and 15-3
  class types and 10-3, 10-5
  classes 10-25
    class methods and 10-28
    constructors and 10-12, 10-13
  comparing 6-9
  components 5-5
  constant-address expressions 5-7
  constructors 10-12
  functions 4-18, 12-3
  global procedures 4-18
  indirect 12-4
  interface objects 13-1, 13-11 to 13-14
    access specifiers and 13-5
    counters 13-11, 13-13
    getting 13-12
    querying 13-13
  memory 20-12, 20-15
  methods 10-10, 13-11
  objects 4-18, 7-9, 10-12
    class methods and 10-28
    class types and 10-2
    Self identifier and 10-3
  ordinal types 4-2
  out parameters 9-8
  packages 15-2, 15-3
  pointer types 4-16, 5-3
  procedures 12-3
  properties 10-20
  strings 4-9, 4-11
  untyped parameters 9-8
  variables 5-2 to 5-3
    as parameters 9-8
    changing types 5-5
  variants 4-21
register expressions 20-12
register operands 20-9
register reserved word 4-18, 9-2, 9-3, 19-1
  Boolean flag parameter and 19-4
  interface objects and 13-5
  parameters and 19-3
  Self parameter and 19-4
registers 9-3, 19-1, 19-3
  assembler and 20-2, 20-8, 20-11, 20-12
  preserving 19-5, 20-2
registration (product) 1-2
relational expressions 6-8 to 6-9
relational operators 6-8
  interface objects and 13-11
  precedence 6-1
  simple expressions 6-3
  variants and 4-25, 4-26
_Release method 13-4, 13-7, 13-11
relocatable expressions (assembler) 20-12
remainders 6-4
Rename procedure 16-1
  strings 17-4
repeat reserved word 7-5, 7-6
repeat statements 7-6
repetitive statements 7-5
requires reserved word 15-2, 15-3
reserved characters 2-2
reserved symbols (assembler) 20-8
  types 20-14
reserved words 2-2 to 2-3
  See also specific
  built-in assembler 20-5
  separators and 2-1
  table of 2-2
Reset procedure 16-1, 16-2, 16-4
  device drivers 16-5
resource strings 3-2, 8-2
resources
  exception handling 11-7
  sharing 14-1
resourcestring reserved word 8-2
@Result symbol (assembler) 20-9
Result variable 9-6
RET instructions 20-3
return characters 2-1
return values
  binary expressions 6-4
  character pointers 6-7
  functions 7-1, 9-5, 9-6, 19-3
    built-in assembler 20-17
    undefined 9-5
  logical expressions 6-5, 6-6
  ordinality 4-2, 4-4
    Boolean values 4-5
  passing 9-8
  relational expressions 6-8
  short strings 4-8
  string expressions 6-6
  unary expressions 6-4
  value typecasts 6-12
  variants 4-21
    caution 4-22
Rewrite procedure 16-1, 16-2, 16-4
  device drivers 16-5
right brace (}) symbol 2-2
right bracket (]) symbol 2-2
RmDir procedure 16-1

Round function 3-2
rounding errors 4-7
routines
  calling in DLLs 9-3
  file I/O 16-3, 16-4
  optimizing 14-1
  storing data 18-8
running programs 7-2
$RUNONLY compiler directive 15-4
runtime errors 11-1, 19-6
  DLLs 14-8
  exceptions failing 11-4
runtime packages 15-1, 15-4
  generating 15-4
runtime type information (RTTI) 10-6
  checking 10-15
  published properties 10-24
runtime values 4-20

S

$S compiler directive 5-2
safecall reserved word 9-2, 19-1
  Boolean flag parameter and 19-4
  interface objects and 13-5, 13-16
  Self parameter and 19-4
saving
  packages 15-1
  property settings 10-24
SBCS (single-byte character sets) 17-6
scale factor syntax diagram 2-4
scope 8-3 to 8-4
  built-in assembler 20-11
  class types 10-4, 10-5
  exceptions 11-5
  interface objects 13-5, 13-11, 13-13
  properties 10-21, 10-25
  type declarations 4-1
    strings 4-9, 4-11
  variables 5-1
  within methods 10-11
Seek procedure 16-2
SeekEof function 16-2
  device drivers 16-5
SeekEoln function 16-2
  device drivers 16-5
segment addresses 20-2
segment registers 20-9
segments 5-7
  overrides 20-15
  storing values 20-4
Self identifier 10-3, 19-4
  class methods and 10-28
  component designators and 5-5, 10-4
  method calls and 10-10, 10-11
SelLength procedure 4-9
semicolon (;) in code 7-4
separating tokens 2-1
separators, defined 2-1
sequential read/writes 16-2
set constructors 6-11
set expressions 6-7
set operators 6-7
set types 4-15, 10-6
  constants as 5-10
  properties and 10-24
  return values 19-3
  setting values 6-11
  storage 18-6
SetMemoryManager procedure 18-2
sets 4-2, 4-5, 4-15
  as parameters 19-2
  comparing 6-9
  empty 6-11
  maximum elements 18-6
  setting values 5-10
  testing membership 6-9
setters 13-3
SetTextBuf procedure 16-2
Shared Memory Manager 14-9
ShareMem unit 14-9
shl operator 6-5
  variant expressions 4-25
short string types 4-7, 4-8
  arrays and 5-4
  as parameters 19-2
  constants as 5-8
  storage 18-5
short-circuit evaluation 6-6
Shortint type 4-3
  internal formats 18-2
ShortString function 4-11
ShortString type 4-8
  pointers 4-17
shr operator 6-5
  variant expressions 4-25
sign identity/negation 6-4
signed numbers 2-5
  valid ranges 4-3
sign-extended values 6-12
significand 18-3
simple expressions 6-3
simple statements 7-1 to 7-3
simple types 4-2 to 4-7
  comparing 6-8
  constants as 5-7
single quote (') in code 2-5
  as literal character 2-5
Single type 4-7
  internal formats 18-3
  simple types and 4-2
single-byte character sets 17-6
SizeOf function 4-4
  constants and 3-2
  open arrays 9-10
small set types 10-6
  properties and 10-24
Smallint type 4-3
  internal formats 18-2
source code
  assembler optimizations 20-16
  exporting 14-1, 14-5, 14-6, 14-9
  importing 14-2 to 14-5
    by name 14-2
    by ordinals 14-2
  testing execution 14-8
source files
  packages 15-1, 15-2
    listed 15-4
space characters 2-1
stack 5-2, 19-1
  large arrays and 9-10
  overflow checking 5-2
  passing parameters and 19-2
  pointers 20-2
  removing parameters 9-2
  unwinding 11-4
stack-based variables 5-7
standard directives 2-3
statements 7-1, 8-1
  exception 7-10
  in blocks 8-2
  simple 7-1 to 7-3
  structured 7-3 to 7-10
static imports 14-4
static linking 14-1
static methods 10-8
static text strings 4-7
status bits 18-1
status information 18-2
status variables 18-2
stdcall reserved word 9-2, 19-1
  Boolean flag parameter and 19-4
  DLLs and 14-5

interface objects and 13-5, 13-15
Self parameter and 19-4
storage
arrays 4-17, 18-6
Boolean types 4-5
character types 18-2
class types 18-9
class-reference types 18-10
compressing 4-12
enumerated types 18-3
file types 18-7
floating-point types 18-3 to 18-4
integer types 18-2
null-terminated strings 17-1
objects 10-12, 10-14
pointer types 18-5
procedural types 18-8
real types 4-7
record types 18-7
set types 18-6
string types 4-8, 4-9, 18-5 to 18-6
variants 4-21, 18-10 to 18-11
storage specifiers 10-24
inherited properties and 10-25
stored reserved word 10-24
property overrides and 10-25
Str procedure 17-4
StrAlloc function 17-5
StrBufSize function 17-5
StrCat function 17-5
StrComp function 17-5
StrCopy function 17-5
StrDispose function 17-5
StrECopy function 17-5
StrEnd function 17-5
StrFmt function 17-5
StrIComp function 17-5
string constants 3-1
built-in assembler 20-8
DLLs 14-7
interface definitions 13-4
memory management 18-5
open arrays 9-10
specific characters 4-4
string literals 17-2
tokens and 2-1
string expressions 6-6, 17-5
external directives 14-2
string functions 9-12, 17-5
wide character conversions 17-7
string lists 11-6
string operator 6-6
string pointers
string constants 18-5
wide strings 18-6
string reserved word 4-8, 4-9
string types 4-7 to 4-11
backward compatible 4-7
case statements and 7-4
constants as 5-8, 5-10
conversions 4-7, 4-11
packed 4-13, 5-8
records and 4-14
return values 19-3
storage 18-5 to 18-6
String variant type 4-25, 4-26
strings 2-5 to 2-6, 4-7, 4-11
assembly language 20-8
comparing 6-8, 17-5
concatenating 6-6, 17-5, 17-6
conversions 11-3, 17-5, 17-7
copying 4-9, 17-5
embedding control characters 2-5
empty 4-9, 9-11
formatting 17-5
length 4-8, 4-9, 17-5, 18-5, 18-6
literal 17-2, 17-6
memory management 18-5
null-terminated 4-10, 17-1 to 17-7
OLE and 4-11
pointers 4-17
storing 4-13
open arrays 9-12
typecasting 4-10, 4-11, 17-6
variables and 5-4
variants and 4-20, 4-23, 4-24
StringToWideChar function 17-7
StrLCat function 17-5
StrLComp function 17-5
StrLCopy function 17-5
StrLen function 17-5
StrLFmt function 17-5
StrLIComp function 17-5
StrLower function 17-5
StrMove function 17-5
StrNew function 17-5
StrPCopy function 17-5
StrPLCopy function 17-5
StrPos function 17-5
StrRScan function 17-5
StrScan function 17-5
StrToIntRange function 11-3
STRUC directives 20-1
structure member selector operator 20-11, 20-15
structured statements 7-3 to 7-10
structured types 4-12 to 4-15
constants as 5-8 to 5-10
dereferencing 5-3
string literals and 17-2
value parameters and 9-7
structures 5-8
StrUpper function 17-5
subclasses (interface objects) 13-9
subexpressions 20-15
subrange types 4-6
subroutines 9-2
subset operator 6-8
substrings 17-5
subtraction 6-4
Succ function 4-2
Boolean types and 4-5
constants and 3-2
successors 4-2
Sum function 9-10
superset operator 6-8
Swap function 3-2
symbols 2-1 to 2-2
absolute addresses 20-12
built-in assembler 20-8, 20-9, 20-14
member selector 20-11
user-defined 20-6
syntax diagrams 1-2
blocks 8-1, 8-2
character strings 2-5 to 2-6
class types 10-1
class-reference types 10-26
constants 5-7
array types 5-8
procedural-typed 5-11
record types 5-9
constructors 10-12
destructors 10-14
DLLs 14-5, 14-6
exceptions 11-2, 11-3, 11-7
expressions 6-2, 6-3
address-of operator 6-10
set constructors 6-11

terms 6-3
value typecasts 6-12
external directives 9-4
field designators 5-4
formal parameters 9-6
function calls 6-11
function declarations 9-5
identifiers 2-3
interface objects 13-2, 13-8
automation
dispatch 13-14
labels 2-5
numbers 2-4
open-array constructors 9-11
packages 15-2
procedures 9-1
programs 12-1
statements 7-1
compound 7-3
conditional 7-3, 7-4
repetitive 7-5, 7-6, 7-7
simple 7-1, 7-2
structured 7-3, 7-9
symbols 2-1
type declarations 4-1
array types 4-12
enumerated types 4-5
file types 4-15
OLE automation
variant 4-29
ordinal types 4-3
pointer types 4-16
procedural types 4-18
record types 4-13, 4-14
set types 4-15
simple types 4-2
structured types 4-12
subrange types 4-6
units 12-2, 12-3
finalization 12-4
implementation 12-3
initialization 12-3
variables 5-1, 5-3, 5-5
qualified names and 5-3
strings as 5-4
system files 16-3
System unit 4-20, 8-4, 12-2
class types 10-28
class-reference types 10-28
I/O procedures 16-1
interface objects 13-4, 13-6
identifier type IDs 13-4
memory procedures 18-2
pointer types 4-17
string functions 9-12, 17-7
value type constants 9-12
variant types 4-22, 4-26, 9-11, 18-10
SysUtils unit
DLLs and 14-9
exception classes 11-1, 11-8
importing 11-3
string functions 17-5

T

$T compiler directive 4-32, 6-10
tag fields 4-14, 5-10
Tan function 11-6
TAutoObject class 10-6
TByteArray type 4-17
TByteRec type 5-6
TClass type 10-27, 10-28
technical support 1-2
terms 6-3
testing for nil values 4-19
testing set membership 6-9
text files 4-15, 16-2, 16-3 to 16-4
assignment 16-2
device drivers 16-4 to 16-6
opening 16-5
storage 18-7
text strings *See* strings
Text type 4-15, 16-3
TextBuf type 4-17
TextFile type 4-15
TGUID type 13-4
the ultimate ancestor 10-4
then reserved word 7-4
time 4-20, 4-23
TInterfacedObject class 13-6, 13-8
to reserved word 7-7
TObject type 10-4, 10-28, 18-9
tokens 2-1
TRectangle type 10-5
Trunc function 3-2
Truncate procedure 16-2
truncation 6-7
short strings 4-8
value typecasts and 6-12
try reserved word 11-3, 11-7
try...except statements 7-10, 11-3 to 11-7
DLLs 14-8
raise keyword and 11-3
terminating 11-4
try...finally statements vs. 11-7
try...finally statements 7-10, 11-7 to 11-8
constructors 10-13
TStringList object 11-6
TTextRec type 4-17, 18-8
Turbo Assembler 20-1
TVarArrayBound type 18-10
TVarData type 18-11
TVarRec type 4-17, 9-11
TWordArray type 4-17
TWordRec type 5-6
type checking (assembler) 20-13 to 20-14
type checking operator 10-15
class-reference types and 10-26
type definitions *See* types, declarations
type reserved word 4-34, 5-1
typecast operator 20-15
typecasting
arrays 4-17
built-in assembler 20-14, 20-15
expressions and 6-12
integers 4-4
interface objects and 13-12
is operator and guarded 10-15
pointer types 4-16
string types 4-10, 4-11, 17-6
untyped parameters and 9-8
variables 5-5 to 5-6
variants 4-22, 4-25
type-checked pointers 4-32
typed files 16-2, 18-7
storage 18-7
type-mismatch errors 10-15, 10-16
types 2-3, 12-3
See also specific types
alignment masks 18-7
binary conversions 4-4
built-in assembler 20-11
class 10-1
class-reference 10-25 to 10-27
comparisons 6-8 to 6-9
compatibility 4-31, 4-32
character strings 17-2, 17-6
class types 10-5
interface objects 13-11

procedural types 4-20
variants 4-22, 4-23
constants as 5-7 to 5-11
declaration part 4-34
declarations 4-1, 4-12, 4-17
automation objects 4-29
in blocks 8-2
in DLLs 14-3
long strings 4-9, 4-10
pointers 4-17
procedural 4-18
record types 4-13
short strings 4-8
exporting 14-1
fundamental vs. generic 4-1
identity 4-31
listed 4-1
private 12-3
properties and 10-19
unknown 10-6, 10-26
polymorphism and 10-27
user-defined 4-5, 4-6
variable 4-20
typography 1-2

U

unary operators 6-4
defined 6-1
logical operations 6-5
variant expressions 4-26
Unassigned constant 4-20
variant expressions 4-25, 4-26
unassigned variables
function calls 4-19
procedures 4-19
unassigned variants 4-20
comparisons 4-25, 4-26
unconditional jump instruction 20-3
undefined characters 4-8
undefined pointers 5-5
undefined values 9-5
underscore characters 2-3
Unicode characters 4-4, 17-6
conversions 4-24
string type 4-8
unions 6-7
unit files 12-2
including in packages 15-2, 15-3
units 2-3, 12-1 to 12-5
built-in assembler 20-11
calling 12-2, 12-4
changing interface part 12-5
circular references 12-5
data processing 12-4
declarations 12-2, 12-3
finalization 12-4
implementation 12-3
initialization 12-3
DLLs vs. 14-1
duplicate identifers 8-4, 12-2
exception handling 11-3
import 14-3
indirect references 12-4
initializing 12-2, 12-3
naming 12-2
scope 8-4
version numbers 12-5
unknown data 4-20
unknown types 10-6, 10-26
polymorphism and 10-27
unpacked record types 18-7
unsigned constants 6-2
unsigned numbers 2-4
valid ranges 4-3
until reserved word 7-6
untyped files 4-15, 16-2, 16-4, 18-7
storage 18-7
untyped parameters 9-8 to 9-9
open arrays and 9-9
variable parameters and 9-8
untyped pointers 4-10, 4-16
assigning values 6-13
uppercase conversions 17-5
uppercase letters 2-2
UserData field 18-8
user-defined identifiers 2-3
user-defined symbols 20-6
user-defined types 4-5, 4-6
uses reserved word 8-4, 12-2, 12-4

V

Val procedure 17-4
value parameters 9-7, 19-2
built-in assembler 20-10
constant parameters vs. 9-7
value type constants 9-12, 9-13
value typecasts 6-12
values
See also nil values; nonzero values; return values
assignment 4-33
Boolean types and 4-6
class types 10-3
class-reference types 10-26
constants 5-7
extended 6-12
ordering string 4-11
passing 9-8
pointers 5-10
predecessors/successors 4-2
procedural 4-19, 4-20
sets 4-2, 4-5, 4-15, 5-10
setting at runtime 4-20
string types and 4-8
structures 5-8
subrange types and 4-6
undefined 9-5
valid ranges
Comp type 18-4
Currency type 18-4
index specifiers 10-23
integer types 4-3
real types 4-7
variables as 5-1
variants 4-20, 4-21
VAnsiString field 9-12
var reserved word 9-7
long strings and 4-10
string types and 4-11
untyped parameters and 9-8
variant arrays and 4-27
varArray bit position 4-22
varArray constant 4-22, 18-10
VarArrayCreate procedure 4-27
VarArrayDimCount function 4-28
VarArrayHighBound function 4-28
VarArrayLoadFile function 4-28
VarArrayLock function 4-28
VarArrayLowBound function 4-28
VarArrayRedim procedure 4-27
VarArrayUnlock procedure 4-28
VarAsType function 4-25
varBoolean constant 4-22, 18-10
varByRef constant 18-10
varByte constant 4-22, 18-10
variant arrays and 4-28
VarCast procedure 4-25
varCurrency constant 4-21, 18-10
varDate constant 4-21, 18-10
varDispatch constant 4-22, 4-25, 13-12, 18-10
varDouble constant 4-21, 18-10
varEmpty constant 4-21, 13-12, 18-10

varError constant 4-22, 18-10
variable parameters 9-8, 19-2, 19-3, 19-4
  built-in assembler 20-10
  out parameters vs. 9-8
variable-length strings 4-7, 4-8
variables 2-3, 12-3
  See also global variables; local variables
  addresses 6-10
  arrays 5-3
  assigning values 7-1, 7-7
  class types 10-2, 10-5
  class-reference types 10-27
  declarations 4-1, 5-1, 8-2
    assembler and 20-4, 20-7, 20-11
    assembly member selector 20-11
    component designators 5-5
    field designators 5-4
    pointers as 5-5
    qualified names and 5-3
  DLLs and 14-3, 14-8
  file 16-2, 16-3, 18-8
  functions as 4-18
  global vs. local 5-1
  interface objects 13-3, 13-11
  local vs. dynamic 5-7
  pointer types and 4-16, 4-17
  pointers as 5-5
  private 12-3
  procedures as 4-18, 4-19, 6-13
  read-only 5-7, 9-7
  record types 5-4
  references 5-2 to 5-3
    changing types 5-5
  scope 5-1
  set types and 4-15
  status 18-2
  string types and 5-4
  subrange types and 4-6
  type identity and 4-31
  typecasting 5-5 to 5-6
  unassigned
    function calls 4-19
    procedures 4-19
    variants 4-20
variable-sized arrays 4-13, 4-27, 9-9
variant operators 6-8
Variant type 4-20, 4-21
  interface objects and 13-12
  internal formats 18-10
  pointers 4-17
  records and 4-14
  return values 19-3
  typecasting 4-22, 4-25
variant types 4-20 to 4-31
  conversions 4-22 to 4-25
  open array 9-11
  storage 18-10 to 18-11
variants
  See also Variant type
  arrays and 4-20, 4-22, 4-26 to 4-28
  as parameters 19-2
  assigning values 4-27
  expressions and 4-22, 4-25 to 4-26
  initializing 4-21
  records and 4-14, 5-10
  referencing automation objects 4-29 to 4-31
  type code 18-11
  type compatibility 4-22, 4-23
  type constants 4-21
  unassigned 4-20
    comparisons 4-25, 4-26
varInteger constant 4-21, 18-10
varNull constant 4-21, 18-10
varOleStr constant 4-22, 18-10
  variant arrays and 4-27
varSingle constant 4-21, 18-10
varSmallint constant 4-21, 18-10
varString constant 4-22, 18-10
  variant arrays and 4-27
VarType function 4-21, 4-22
varTypeMask constant 4-22, 18-10
varUnknown constant 4-22, 13-12, 18-10
varVariant constant 4-22, 18-10
version numbers 12-5
virtual constructors 10-27
virtual memory 18-1
virtual method tables 18-9, 18-10
virtual methods 10-8 to 10-9
  constructors as 10-14
  dynamic vs. 10-9
virtual reserved word 10-8
VirtualAlloc function 18-1
VirtualFree function 18-1
visibility 10-5 to 10-7
  See also scope
VMT (virtual method tables) 18-9, 18-10
v-tables 18-9
  binding 13-16
vtAnsiString constant 9-13
vtBoolean constant 9-13
vtChar constant 9-13
vtClass constant 9-13
vtCurrency constant 9-12, 9-13
vtExtended constant 9-12, 9-13
vtInteger constant 9-13
vtInterface constant 13-14
vtObject constant 9-13
vtPChar constant 9-13
vtPointer constant 9-13
vtPWideChar constant 9-13
vtString constant 9-12, 9-13
vtVariant constant 9-12, 9-13
vtWideChar constant 9-13
VType field 9-12

W

$WEAKPACKAGEUNIT compiler directive 15-4
while reserved word 7-5, 7-6
while statements 7-6
whitespace characters 2-1
whole numbers 4-3
wide character arrays 17-7
wide characters 4-4, 17-6
  pointers 4-17
wide string types 4-8, 4-11
  arrays and 5-4
  storage 18-6
WideChar type 4-4
  internal formats 18-2
WideCharLenToString function 17-7

WideCharToString
function 17-7
WideString type 4-11
Win32 virtual memory 18-1
Windows API routines 9-3
Windows message
handling 10-16
Windows unit 14-3
with reserved word 7-9
with statements 7-9 to 7-10
as operator and 10-16
class types and 5-5, 10-4,
10-10
field designators and 5-4
interface objects and 13-14
method calls 10-11
Word type 4-3
internal formats 18-2
WordBool type 4-3, 4-5
internal formats 18-2
words 17-6, 20-4
accessing 5-6
Boolean types and 4-5
character types and 4-4
operands and 4-4
set types and 10-6
write directives 10-20, 10-21
access specifiers and 10-21,
10-22
index specifiers and 10-23
interface objects 13-5
property overrides
and 10-25
Write procedure 16-2
device drivers 16-5
strings 17-4
text files 16-3
untyped files 16-4
write reserved word 10-21,
10-23, 10-25
Writeln procedure 16-2
device drivers 16-5
strings 17-4
write-only files 16-2
write-only properties 10-21

X

$X compiler directive
character strings and 2-6
variables references and 5-3
xor operator
bitwise operations 6-5
Boolean operations 6-6
variant expressions 4-25

Z

-Z compiler directive 15-5
zero-based character
arrays 4-13, 5-9, 17-1
zero-length strings 4-9
zeros
division errors 11-4
labels and leading 2-5

Borland
www.borland.com

Copyright © 1997 Borland International, Inc. All rights reserved. All Borland product names are trademarks of Borland International, Inc. Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Internet: http://www.borland.com CompuServe: GO BORLAND. Offices in: Australia, Canada, France, Germany, Hong Kong, Japan, Latin America, Mexico, The Netherlands, Taiwan, and United Kingdom • Part # HDA1330WW21771 • BOR 9961